SYBASE®
An SAP® Company

## Design Guide

# Replication Server® 15.7.1 SP100

DOCUMENT ID: DC32580-01-1571100-01
LAST REVISED: May 2013

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

# Contents

# Conventions

These style and syntax conventions are used in Sybase® documentation.

*Style conventions*

| Key | Definition |
| --- | --- |
| `monospaced(fixed-width)` | • SQL and program code<br>• Commands to be entered exactly as shown<br>• File names<br>• Directory names |
| *`italic monospaced`* | In SQL or program code snippets, placeholders for user-specified values (see example below). |
| *italic* | • File and variable names<br>• Cross-references to other topics or documents<br>• In text, placeholders for user-specified values (see example below)<br>• Glossary terms in text |
| **bold san serif** | • Command, function, stored procedure, utility, class, and method names<br>• Glossary entries (in the Glossary)<br>• Menu option paths<br>• In numbered task or procedure steps, user-interface (UI) elements that you click, such as buttons, check boxes, icons, and so on |

If necessary, an explanation for a placeholder (system- or setup-specific values) follows in text. For example:

Run:

```
installation directory\start.bat
```

where *installation directory* is where the application is installed.

*Syntax conventions*

| Key | Definition |
|---|---|
| { } | Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces when you enter the command. |
| [ ] | Brackets mean that choosing one or more of the enclosed options is optional. Do not type the brackets when you enter the command. |
| ( ) | Parentheses are to be typed as part of the command. |
| \| | The vertical bar means you can select only one of the options shown. |
| , | The comma means you can choose as many of the options shown as you like, separating your choices with commas that you type as part of the command. |
| ... | An ellipsis (three dots) means you may repeat the last unit as many times as you need. Do not include ellipses in the command. |

*Case-sensitivity*

- All command syntax and command examples are shown in lowercase. However, replication command names are not case-sensitive. For example, **RA_CONFIG**, **Ra_Config**, and **ra_config** are equivalent.
- Names of configuration parameters are case-sensitive. For example, **Scan_Sleep_Max** is not the same as **scan_sleep_max**, and the former would be interpreted as an invalid parameter name.
- Database object names are not case-sensitive in replication commands. However, to use a mixed-case object name in a replication command (to match a mixed-case object name in the primary database), delimit the object name with quote characters. For example: **pdb_get_tables** "*TableName*"
- Identifiers and character data may be case-sensitive, depending on the sort order that is in effect.
  - If you are using a case-sensitive sort order, such as “binary,” you must enter identifiers and character data with the correct combination of uppercase and lowercase letters.
  - If you are using a sort order that is not case-sensitive, such as “nocase,” you can enter identifiers and character data with any combination of uppercase or lowercase letters.

*Terminology*

Replication Agent™ is a generic term used to describe the Replication Agents for Adaptive Server® Enterprise, Oracle, IBM DB2 UDB, and Microsoft SQL Server. The specific names are:

- RepAgent – Replication Agent thread for Adaptive Server Enterprise
- Replication Agent for Oracle

- Replication Agent for Microsoft SQL Server
- Replication Agent for UDB – for IBM DB2 on Linux, Unix, and Windows
- Replication Agent for DB2 for z/OS

# Introduction

Use Replication Server® to create and maintain distributed data applications.

## Centralized and Distributed Database Systems

Changes in the corporate environment toward decentralized operations have prompted organizations to move toward distributed database systems.

In the traditional enterprise computing model, an Information Systems department use to maintain control of a centralized corporate database system. Mainframe computers, usually located at corporate headquarters, provided the required performance levels. Remote sites used to access the corporate database through Wide Area Networks (WANs) using applications provided by the Information Systems department.

Today's global enterprise may have many Local Area Networks (LANs) joined with a WAN, as well as additional data servers and applications on the LANs. Client applications at the sites need to access data locally through the LAN or remotely through the WAN. For example, a client in Tokyo might locally access a table stored on the Tokyo data server or remotely access a table stored on the New York data server.

In a distributed database environment, mainframe computers may be needed at corporate or regional headquarters to maintain sensitive corporate data, while clients at remote sites use minicomputers and server-class workstations for local processing.

Both centralized and distributed database systems must deal with the problems associated with remote access:

- Network response slows when WAN traffic is heavy. For example, a mission-critical transaction-processing application may be adversely affected when a decision-support application requests a large number of rows.
- A centralized data server can become a bottleneck as a large user community contends for access.
- Data is unavailable when a failure occurs on the network.

## Advantages of Replicating Data

The performance and availability problems associated with remote database access can be solved by replicating the data from its source database to a local database. Replication Server provides a cost-effective, fault-tolerant system for replicating data.

Replication Server keeps data up to date in multiple databases so that clients can access local data instead of remote, centralized databases. Compared to a centralized data system, a

replication system provides improved system performance and data availability and reduces communication overhead.

Because it transfers transactions, not rows, Replication Server maintains the integrity of replicated data across the system, while also increasing data availability. Replication Server also allows you to replicate stored procedure invocations, further enhancing performance.

### *Improved Performance*

In a distributed replication system, data requests are completed on the local data server without the client having to access the WAN. Performance for local clients is improved because:

- LAN data transfer rates are faster than WAN data transfer rates.
- Local clients share local data server resources instead of competing for central data server resources.
- Traffic and contention for locks are considerably reduced because local decision-support applications are separated from centralized OLTP applications.

### *Greater Data Availability*

In a distributed replication system, data is replicated at local and remote sites, so clients can continue to work regardless of what happens at the primary data source or over the WAN.

- When a failure occurs at a remote site, clients can continue to use local copies of replicated data.
- When a WAN failure occurs, clients can continue to use local replicated data.
- When the local data server fails, clients can switch to replicated data at another site.

When WAN communications fail, Replication Servers at other sites store transactions in stable queues (disk storage) so that replicated tables at the unavailable site can be brought up to date when communications resume. When a replicated function is initiated in a source database, it is stored in stable queues until it can be delivered to the destination site.

## Data Distribution with Replication Server

Replication Server replicates transactions—incremental changes instead of data copies—and stored procedure invocations, rather than the stored procedures themselves, providing a high-performance distributed data environment while maintaining data integrity

Replication Server distributes data over a network by:

- Providing application developers and system administrators with a flexible publish-and-subscribe model for marking data and stored procedures to be replicated
- Managing replicated transactions while retaining transaction integrity across the network

## Publish-and-Subscribe Model

Replication Server uses a publish-and-subscribe model for data replication between primary and replicate databases.

In a Replication Server system, transactions in a source database are detected by a Replication Agent and transferred to the local Replication Server, which distributes the information across LANs and WANs to Replication Servers at destination sites. Those Replication Servers update the target database according to the requirements of the remote client. If a network or system component fails, data in the process of being delivered is temporarily stored in queues. When the failed component returns to operation, the replication system resynchronizes copies of the data and normal replication resumes.

The primary data is the source of the data that Replication Server replicates in other databases. You "publish" data at primary sites to which Replication Servers at other sites "subscribe." You first create a replication definition to designate the location of the primary data. The replication definition describes the table structure and names the database that contains the primary copy of the table. For easier management, you may collect replication definitions into publications.

The creation of a replication definition or publication does not, by itself, cause Replication Server to replicate data. You must create a subscription against the replication definition (or publication) to instruct Replication Server to replicate the data in another database. A subscription resembles a SQL **select** statement. It can include a **where** clause to specify which rows of a table you want to replicate in the local database, allowing you to replicate only the necessary data.

Beginning with the 11.5 version of Replication Server, you can have multiple replication definitions for a primary table. Replicate tables can subscribe to different replication definitions to obtain different views of the data.

Once you have created subscriptions to replication definitions or publications, Replication Server replicates transactions to databases with subscriptions for the data.

## Replicated Functions

Replicating Adaptive Server stored procedures asynchronously between databases can improve performance over normal data replication by encapsulating many changes in a single replicated function. Because they are not associated with table replication definitions, replicated functions can execute stored procedures that may or may not modify data directly.

You can replicate stored procedure invocations from a primary database to a replicate database, or from a replicate database to a primary database.

With replicated functions, you can execute a stored procedure in another database. You can:

- Replicate the execution of an Adaptive Server stored procedure to subscribing sites

- Improve performance by replicating only the name and parameters of the stored procedure rather than the actual changes

Like tables, replicated stored procedures may have replication definitions, which are called function replication definitions, and subscriptions. When a replicated stored procedure executes, the Replication Server passes its name and execution parameters to subscribing sites, where the corresponding stored procedure executes.

You create function replication definitions at the primary data site. Replication Server supports applied functions and request functions.

An applied function and a request function are replicated from a primary to a replicate database. You create subscriptions at replicate sites for the function replication definition and mark the stored procedure for replication in the primary database. The applied function is applied at replicate database by maint_user whereas the request function is applied at replicate database by the same user who executes the stored procedure at the primary database.

### See also


- *Applied Functions* on page 40
- *Request Functions* on page 64


## Transaction Management

Replication Server depends on data servers to provide transaction-processing services. To guarantee the integrity of distributed data, data servers must comply with transaction-processing conventions, such as atomicity and consistency.

Data servers that store primary data provide most of the concurrency control needed for the distributed database system. If a transaction fails to update a table with primary data, Replication Server does not distribute the transaction to other sites. When a transaction does update primary data, Replication Server distributes the changes and, unless a failure occurs, the update succeeds at all sites that have subscribed to the data.

To maintain replicated data consistency, Replication Server uses optimistic concurrency control, which:

- Promotes high availability of data because it does not lock the data for the duration of the distributed transaction. Instead it rolls back changes in the event of a conflict.
- Requires fewer system resources to process a transaction.
- Does not require data servers to have special distributed transaction processing features in order to participate in a distributed transaction.

#### *Failed Replicated Transactions*

A modification to primary data may fail to update a replicate copy of the data at another site.

Some reasons for the failure of an update to a replicated table are:

- The data server's maintenance user login name does not have the permissions needed to update the replicate data.
- The replicate and primary versions of the data are inconsistent after a system recovery.
- A client updates replicate data directly rather than updating the primary version.
- The data server storing the replicate table has constraints that are not enforced by the data server storing the primary version.
- The data server storing the replicated copy of the table rejects the transaction due to a system failure, such as lack of space in the database.

When a transaction fails, Replication Server receives an error from the data server. Data server errors are mapped to Replication Server error actions. The default action for a failed transaction is to write a message in the Replication Server error log (including the message returned by the data server) and then suspend the database connection. After you correct the cause of the failure, you can resume the database connection and Replication Server will retry the failed transaction.

You also can have Replication Server record a failed transaction in the exceptions log (a set of three tables in the Replication Server System Database) and continue processing the next transaction. See Replication Server for a description of the RSSD.

If you use the exceptions log, you must manually resolve the transactions that are saved there to make the replicate data consistent with the primary data. In some cases, the process can be automatic by encapsulating the logic for handling the rejected transactions in an intelligent application program.

#### *Transactions That Modify Data in Multiple Data Servers and Databases*

A transaction that modifies primary data in more than one data server may require additional concurrency control. According to the transaction processing requirements, either all of the operations in the transaction are performed or none of them are performed. If a transaction fails on one data server, it must be rolled back on all other data servers updated in the transaction.

Normally, there is exactly one Replication Agent for each primary database. If a single transaction updates multiple primary databases, that transaction is replicated as multiple independent transactions, one for each primary database. Or, you can select to encapsulate such transactions in a single stored procedure, which then flows as an atomic unit to subscribing sites.

### See also

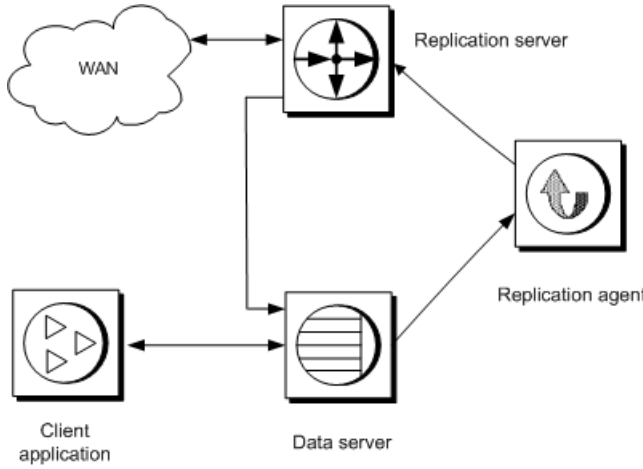
- *Replication Server* on page 10

# Replication System Components

Replication Server has an open architecture that allows you to build a replication system from existing systems and applications and add to it as your organization grows and changes.

One replication system site in a WAN-based, distributed database system that uses Replication Server.

**Figure 1: Replication System**




## Replication System Domain

A Replication system domain refers to all replication system components that use the same ID Server.

You can set up multiple replication system domains, with these restrictions:

- Replication Servers in different domains cannot exchange data. Each domain must be treated as a separate replication system with no cross-communication between them. You cannot create a route between Replication Servers in different domains.
- A database can be managed by only one Replication Server in one domain. Any given database is in one, and only one, ID Server's domain. This means you cannot create multiple connections to the same database from different domains.

## Replication Server

A Replication Server at each site coordinates data replication activities for local data servers, and exchanges data with Replication Servers at other sites.

A Replication Server:

- Receives primary data transactions from databases via Replication Agents and distributes them to sites with subscriptions for the data
- Receives transactions from other Replication Servers and applies them to local databases

Replication Server system tables store information needed to accomplish these tasks. The system tables include descriptions of replication definitions and subscriptions, security records for Replication Server users, routing information for other sites, access methods for the local databases, and other administrative information.

Replication Server system tables are stored in an Adaptive Server database called the Replication Server System Database (RSSD), or an SQL Anywhere® (SA) database called the Embedded Replication Server System Database (ERSSD). An RSSD or ERSSD is assigned to each Replication Server. An Adaptive Server data server with an RSSD can also store application databases. For more information, see the *Replication Server Administration Guide Volume 1*.

Use Replication Command Language (RCL) or the Replication Manager plug-in of Sybase Central™ to manage information in Replication Server. You can execute RCL commands, which resemble SQL commands, on Replication Server using **isql**, the Sybase interactive SQL utility. The *Replication Server Reference Manual* is the complete reference for RCL. You can find information about Replication Manager and Replication Monitoring Services in the *Replication Server Administration Guide Volume 1* and in the online help for Replication Manager.

### See also

- *Capacity Planning* on page 111

## Partitions and Stable Queues

Replication Server stores messages on disk to ensure they can be delivered following a failure.

When you install a Replication Server, you allocate an initial disk partition that Replication Server uses for disk storage. You can add partitions when you have finished installing the Replication Server.

The partition is either a raw disk device or operating system file. Because UNIX operating systems buffer file I/O, you may not be able to completely recover data following a failure. On such a system, use operating system files for partitions only in a test environment. Use raw disk partitions for production environments. See the *Replication Server Reference Manual* for more information about adding partitions.

Replication Server allocates stable queues from its disk partitions for the routes and connections it serves. Messages are saved in the stable queues at least until the messages are confirmed as received at their destination.

The amount of disk space you should allocate for Replication Server partitions depends on the size of transactions and the transaction rate for your application. Stable queues act as buffers for data as it flows through your replication system. If a remote site's Replication Server is unreachable during a network failure, the primary Replication Server stores transactions in a

stable queue until communication is restored. The more space allocated for disk partitions, the longer the Replication Server can queue data without interrupting operations in the primary database.

## ID Server

The ID Server is a Replication Server that registers all Replication Servers and databases in the replication system.

The ID Server must be running each time you:

- Install Replication Server
- Create route
- Create or drop database connection

Because of this requirement, the ID Server is the first Replication Server that you start when you install a replication system.

The ID Server must have a login name for Replication Servers to use when they connect to the ID Server. The login name is recorded in the configuration files of all Replication Servers in the replication system by the **rs_init** configuration program.

## Replication Environment

A replication environment consists of a set of servers that participate in replication. This includes the data servers, Replication Agents, Replication Servers, and DirectConnect™ servers. A replication environment does not have to contain all servers in a replication system domain.

## Replication Manager

The Replication Manager (RM) is installed as a plug-in to Sybase Central. RM is a management utility for developing, managing, and monitoring replication environments.

See the *Replication Server Administration Guide Volume 1.*

## Replication Monitoring Services

The Replication Monitoring Services (RMS) acts as the middle tier in a three-tier management solution for a replication environment.

RMS monitors the health of the servers and components in the replication environment and provides information to troubleshoot problems and commands to fix the problems. See the *Replication Server Administration Guide Volume 1.*

## Data Servers

Data servers manage databases containing primary or replicated data. Client applications use data servers to store and retrieve data and to process queries and transactions. Replication Server maintains replicated data in data servers by logging in as a database user.

Replication Server supports heterogeneous data servers through an open interface. Any system for storing data can be used as a data server if it supports a set of required data operations and transaction processing directives.

### See also

- *Non-ASE Data Server Support* on page 18

## Replication Agent

A Replication Agent transfers transaction log information, which represents changes made to primary data, from a data server to a Replication Server for distribution to other databases.

The Replication Agent reads the database transaction log and transfers log records for replicated tables and replicated stored procedures to the Replication Server that manages the database. The Replication Server reconstructs the transaction and forwards it to the sites that have subscriptions for the data.

A Replication Agent is needed for each database that contains primary data or for each database where replicated stored procedures are executed. A database that contains only copies of replicated data and has no replicated stored procedures does not require a Replication Agent.

Because RepAgent is an Adaptive Server thread and not a separate process, and because most of the system diagrams involve Adaptive Server and not any of the other non-Sybase supported databases, Replication Agent is not shown as a separate icon.

## Client Applications

A client application is a program that accesses a data server. If you use Adaptive Server as the data server, you can create client applications using Open Client/Server™, Embedded SQL™, PowerBuilder®, or any other front-end development tool that is compatible with Sybase Client/Server Interfaces™ (C/SI).

Client applications should update only the primary data. Replication Server distributes the changes to the other sites. Client applications that do not modify data do not need to distinguish between primary and replicated data.

# Replication Management Solutions

Replication Server offers two-tier and three-tier management solutions to support different replication environments.

## Two-tier Management Solution

In a two-tier management solution, RM manages the replication environment by connecting directly to servers in the environment without communicating through a management layer.

This two-tier management solution lets you manage small, simple replication environments with fewer than ten servers. You can create, alter, and delete components in the replication environment. In addition to managing the replication environment, RM also lets you monitor the status of the servers and replication components in the replication environment.

## Three-tier Management Solution

In a three-tier management solution, RM can manage larger and complex replication environments with the help of RMS. RM connects to the servers in the environment through RMS.

RMS provides the monitoring capabilities for the replication environment. In this three-tier management solution, RMS monitors the status of the servers and other components in the replication environment, and RM provides the client interface that displays the information provided by RMS.

# Replication System Components

Replication Server, Replication Agent, and Adaptive Server use C/SI to communicate over a network. In addition, Replication Server uses routes and connections to send messages to other Replication Servers and databases.

## Interfaces File

Server programs such as data servers, Replication Servers, and Replication Agents are registered in an interfaces file (`sql.ini` in Windows and `interfaces` in UNIX) or a Lightweight Directory Access Protocol (LDAP) server so that client applications and other server programs can locate them.

Generally, one interfaces file at each site contains entries for all of the local and remote Replication Servers and data servers. The entry for each server includes its unique name and the network information that other servers and client programs need to connect to it.

Use a text editor to maintain your interfaces file. For information about LDAP Servers, see the *Replication Server Administration Guide Volume 1*.

**Note:** If you are using network-based security, available with Replication Server version 12.0 and later, use the directory services of your network security mechanism (rather than the interfaces file) to register Replication Servers, Adaptive Servers, and gateway software. Refer to the documentation that comes with your network-based security mechanism for details. See *Replication Server Administration Guide Volume 1 > Managing Replication Server Security* for information on how to manage network-based security.

## Routes and Connections

Routes and connections allow Replication Servers to send messages to each other and to send commands to databases.

A route is a one-way message stream that sends requests from one Replication Server to another Replication Server. A connection is a message stream from a Replication Server to a database. Replication Server uses a logical connection to represent the active and standby databases in a warm standby application.

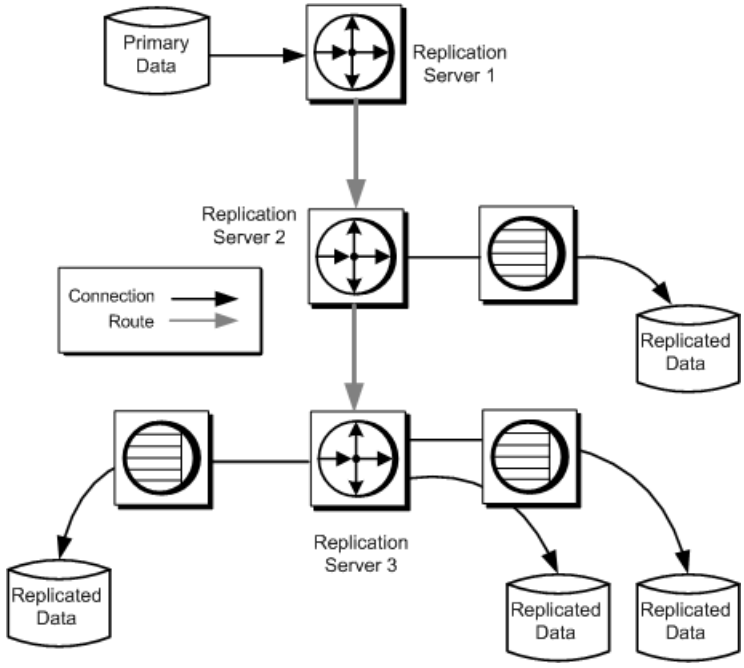
To replicate data from one database into another, you must first establish the routes and connections that allow Replication Server to move the data from its source to its destination.

When you add a database to your replication system, Sybase Central or **rs_init** creates the connection for you. You never have to create a connection directly unless you are replicating data into a database that is not an Adaptive Server.

If you have more than one Replication Server in your replication system, you must create routes between them. If you have only one Replication Server, you do not need to create routes.

This figure illustrates connections and routes between three Replication Servers, one database storing primary data, and four databases storing replicated data.

**Figure 2: Routes and Connections**




When you create a route from a primary Replication Server to a replicate Replication Server, transactions flow from the primary server to the replicate server.

If you plan to execute replicated stored procedures in a replicate database to update a primary database, you must also create a route from the replicate Replication Server to the primary Replication Server.

### Direct and Indirect Routes

In a replication system that has one primary Replication Server and many replicate Replication Servers, you can use indirect routes to reduce the load on the primary Replication Server. Indirect routes allow Replication Server to send messages to multiple destinations through a single intermediate Replication Server.

Routes with intermediate sites have important advantages:

- Reduced WAN volume

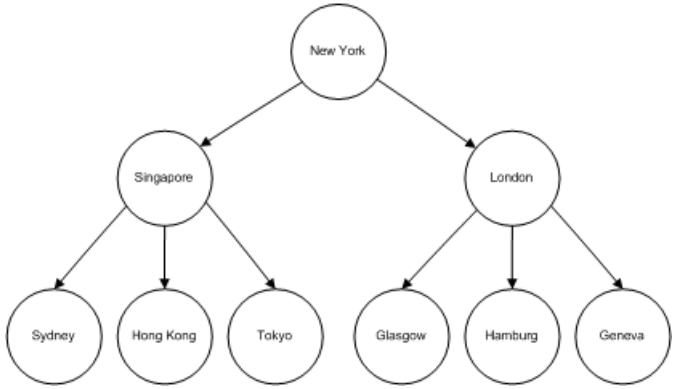
  Replication Server distributes one copy of a message to each intermediate site. Replication Servers at the intermediate sites duplicate the messages for each of their outgoing queues.
- Reduced Replication Server load

  Additional Replication Servers running on separate computers share the processing load, reducing the amount of processing required of Replication Servers at primary sites.
- Fault tolerance

Messages stored at intermediate sites can be used to recover from partition failures at remote sites. See the *Replication Server Administration Guide Volume 1* for details.

This figure shows how message distribution is handled using intermediate sites. The message follows a direct route to the intermediate sites. From the intermediate site, it follows a direct route to the local site. With this routing arrangement, the primary site sends two messages rather than eight.

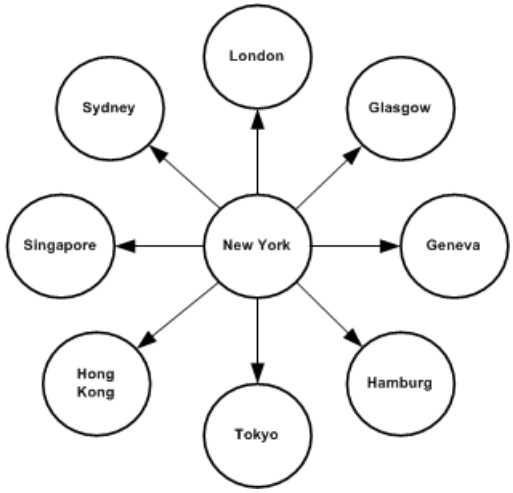
**Figure 3: Routes in a Hierarchical Configuration**




Intermediate sites reduce primary site message volume, but they increase the time between updates at the primary and replicate servers. Plan your routes carefully; use only the number of intermediate sites required.

If you do not use intermediate sites, routes are set up in a star configuration.

**Figure 4: Direct Routes in a Star Configuration**



When a row is updated at a primary site, the primary Replication Server sends messages through the WAN to each remote site that has a subscription for the row. For example, in the above figure, New York can send identical data through eight different routes. If there are many sites, the network is quickly overloaded with redundant messages.

Creating routes in a hierarchical arrangement allows load balancing by reducing the number of connections and messages distributed from the primary site. Additional Replication Servers running on separate computers share the processing load, reducing the amount of processing required of Replication Servers at primary sites.

## Master Database Replication

The master database controls the operation of Adaptive Server and stores information about every user database and associated database devices.

You can replicate the master database, although only the Data Definition Language (DDL) and system commands used to manage logins and roles are replicated. Master database replication does not replicate data from system tables, data or procedures from any other user tables in the master database.

Both the source Adaptive Server and the target Adaptive Server must have the same hardware architecture type (32-bit versions and 64-bit versions are compatible) and the same operating system (different versions are also compatible).

For a list of supported DDL and system procedures that apply to master database, see the *Replication Server Administration Guide Volume 2*.

Replication Server 12.0 and later supports master database replication with warm standby, and with MultiSite Availability (MSA) in Replication Server 12.6 and later. The primary or active Adaptive Server must be Adaptive Server 15.0 ESD #2 and later.

See the *Replication Server Administration Guide Volume 1* for information about master database replication in MSA, and the *Replication Server Administration Guide Volume 2* for information about master database replication in a warm standby environment.

## Non-ASE Data Server Support

The open architecture of Replication Server supports non-ASE data servers in replication systems.

The open architecture includes:

- Sybase Enterprise Connect™ Data Access
- ExpressConnect for HANA DB
- ExpressConnect for Oracle
- Replication Agents

- Error classes and error processing actions
- Functions, function strings, and function-string classes
- User defined datatypes (UDD) and datatype translations
- Connection profiles

See the *Replication Server Heterogeneous Replication Guide* and Replicating Data Into Non-Adaptive Server Data Servers for more information.

**See also**

- *Data Replication into Non-Adaptive Server Data Servers* on page 91

## Enterprise Connect Data Access (ECDA)

ECDA is an integrated set of software applications and connectivity tools that allows you to access data within a heterogeneous database environment.

ECDA lets you access a variety of LAN-based, non-Sybase data sources, as well as mainframe data sources. It consists of components that provide transparent data access within an enterprise. You require a specific ECDA component for each actively supported non-ASE database. See the *Replication Server Options Overview Guide*.

## ExpressConnect for Oracle

ExpressConnect for Oracle (ECO), which is available with Replication Server Options version 15.5 and later, provides direct communication between Replication Server and a replicate Oracle data server.

ExpressConnect for Oracle eliminates the need for installing and setting up a separate gateway server, thereby improving performance and reducing the complexities of managing a replication system. See *ExpressConnect for Oracle Installation and Configuration Guide* for detailed information on ECO.

## ExpressConnect for HANA DB

ExpressConnect for HANA DB is an embedded library loaded by Replication Server for replication to HANA DB.

ExpressConnect for HANA DB is installed with Replication Server 15.7.1 SP100 to provide direct communication between Replication Server and a replicate HANA DB data server.

ExpressConnect for HANA DB is installed with Replication Server. There is no separate installer for ExpressConnect for HANA DB.

See the *Replication Server Heterogeneous Replication Guide > HANA DB as Replicate Data Server > ExpressConnect for HANA DB and Replicate Database Connectivity for HANA DB*.

## Replication Agents

A Replication Agent is required for every database that stores primary data or initiates replicated functions.

A Replication Agent reads the data server transaction log to detect changes to primary data and executions of replicated stored procedures. The transaction log provides a reliable source of information about primary data modifications because it contains records of committed, recoverable transactions.

## Processing Data Server Errors

Replication Server processes the errors and status codes returned by data servers according to your instructions. Each vendor's data server has a different set of error codes.

Replication Command Language (RCL) commands allow you to:

- Create an error class to group together the error code mappings for a database.
- Assign error actions, such as **warn**, **retry_log**, and **stop_replication**, to data server error codes.
- Associate an error class with a database.

**Note:** Replication Server 15.2 and later includes for actively supported databases, pre-loaded error classes with associated error actions. See *Replication Server Administration Guide Volume 1 > Connection Profiles*.

### See also

- *Error Class* on page 95

## Functions, Function Strings, and Function-String Classes

A function is a Replication Server object that represents a data server operation, such as **insert**, **delete**, and **begin transaction**.

In order to operate in a heterogeneous database environment, Replication Server differentiates database commands from the functions it uses to distribute data server requests to other sites. Replication Server uses function strings to convert functions into data-server-specific commands. A function string is a template that Replication Server uses to generate a command the data server can interpret as a transaction-control directive or data-modification instruction.

A function-string class is the set of all function strings used with a database. Function-string classes are provided for Adaptive Server and DB2 data servers. Function strings for transaction control directives are defined just once for each function-string class. Function strings to insert a row, delete a row, or update a row are defined once for each replicated table in a database.

A function string can contain variables—identifiers enclosed in question marks (?)—that represent the values of columns, procedure parameters, system-defined information, and user-

defined variables. Replication Server replaces the variables with actual values before sending the function strings to the data server.

Function strings can be used to generate either Remote Procedure Call (RPC) or database commands such as SQL statements, depending on their format. An RPC-formatted function string contains a remote procedure call followed by a list of data parameters. Embedded variables can be used to assign runtime values to the parameters. Replication Server interprets RPC function strings, builds a remote procedure call, and replaces the variables with runtime data values. An RPC can execute a registered procedure in an Open Server™ gateway to a data server or a stored procedure in an Adaptive Server.

A language-formatted function string passes a database command to the data server. Replication Server does not attempt to interpret the string, except to replace embedded variables with runtime data values. For example, several relational database servers use the SQL database language. A SQL command should be represented as a language function string.

RPC function strings can be more efficient than language function strings because the network packets sent from Replication Server are more compact.

For example, use the **create connection with using profile** clause to create a connection to a DB2 replicate database using a specific version of the profile—v9_1. In this example, the command overrides the command batch size provided by the connection profile with a new value—16384:

```
create connection to db2.subsys
using profile rs_ase_to_db2;v9_1
set username to db2_maint
set password to db2_maint_pwd
set dsi_cmd_batch_size to '16384'
```

**Note:** Replication Server 15.2 and later includes function-string classes pre-loaded with function strings for actively supported databases. See *Replication Server Administration Guide Volume 1 > Connection Profiles*.

## Replication Server Security

Replication Server security includes password-protected login names and a permission system based on the **grant** and **revoke** commands.

Replication Server 12.0 and later supports third-party security services that ensure secure message transmission over the network and that enable user authentication. Replication Server 12.5 and later supports secure socket layer (SSL) session-based security through the Advanced Security option.

## Login Names

Each Replication Server uses login names that are distinct from data server login names. Many clients do not need a Replication Server login name because they accomplish their work through data server applications.

### Replication Server Login Names

When you install a Replication Server, **rs_init** creates Replication Server login names that other Replication Servers and Replication Agents use to log in to Replication Server.

A replication system administrator creates and manages the Replication Server login names and passwords used to manage replicated data or replication system functions such as the addition of new users or a route change. Passwords can be encrypted.

### Data Server Login Names

Data server login names are used with a client application to connect to a data server. A database administrator creates and manages data server login accounts.

Client access to replicated copies of tables is also managed by the Database Administrator. Since Sybase recommends that replicated tables be read-only, clients may be permitted to view replicated data but should be prevented from inserting, deleting, or updating rows.

To modify a table, a client must have a login name on the data server where the primary data is stored as well as the permissions necessary to update the primary data.

To modify replicated tables, a client must modify the primary data so that the Replication Server can distribute the changes to replicate databases that have subscriptions for the data. To modify a table, a client must have a login name on the data server where the primary data is stored as well as the permissions necessary to update the primary data.

### Data Server Maintenance User Login Name

Replication Server uses a maintenance user login name for each local data server database that contains replicated tables. Replication Server uses this login to maintain replicated tables.

The Database Administrator must make sure that the maintenance user login name has the permissions needed to update the replicated tables in the database.

Normally, transactions applied by the maintenance user are filtered by the Replication Agent so they are not replicated out of a database. In certain applications, however, these transactions must be replicated.

#### See also

- *Implementation Strategies* on page 37

## Permissions

Learn about granting and revoking Replication Server permissions for clients.

**Table 1. Replication Server Permissions**

| Permission | Capabilities |
|---|---|
| **sa** | Gives recipient system administrator capabilities. Clients with **sa** permission can perform any Replication Server command. |
| **create object** | Allows recipient to create, alter, or drop Replication Server objects, including replication definitions and subscriptions. |
| **primary subscribe** | Gives the recipient permission to create a subscription in a primary database, but not to create other objects. To create a subscription at a remote site, a client needs **create object** permission in the replicate database and **create object** or **primary subscribe** permission in the primary database. |
| **connect source** | This permission is required for login names used by the Replication Agent. It allows the recipient to execute the RCL commands that are reserved for Replication Agents. |

## Network-Based Security

You can use a third-party network-based security mechanism to authenticate users at login.

Authentication is the process of verifying that users are who they say they are. Users receive a credential that can be presented to remote servers, allowing them seamless access to the components of the replication system through a single login.

Network-based security mechanisms also provide a variety of data-protection services, such as message confidentiality and out-of-sequence checking. Replication Server requests the service, prepares the data, and sends it to the network server for encryption or validation. Once the service is performed, data is returned to the requesting Replication Server for distribution.

Once a secure pathway has been established, data can move in both directions. Both ends of the pathway must support the same security mechanism and be configured the same. The security mechanism must be installed on all machines that use network security, and Replication Server 12.0 or later must be installed on all participating machines.

See the *Replication Server Reference Manual* for information about network-based security in the replication system.

## Advanced Security Option

The Replication Server Advanced Security option provides Secure Socket Layer (SSL), session-based security. SSL is the standard for securing the transmission of sensitive information over the Internet.

SSL provides a lightweight, easy-to-administer security mechanism with several encryption algorithms. It is intended for use over those database connections and routes where heightened security is required.

See the Replication Server *Administration Guide Volume 1* for information about using the Advanced Security option.

# Summary

A quick glance about key Replication Server concepts.

- Replication Server maintains copies of tables in different databases on a network. The replicated copies provide two advantages to users at the sites: faster response and greater availability.
- A replication system built on Replication Server uses ExpressConnect or ECDA gateways to connect to heterogeneous replicate data servers like HANA, Oracle, and Microsoft SQL Server.
- Replication Server is designed with an open interface that allows non-Sybase data servers to be included in a replication system.
- One copy of a table is the primary version. All others are replicated copies.
- Using subscriptions, a replicated copy of a table may contain a subset of the rows in a table.
- Replication Server security consists of login names, passwords, and permissions. Replication Server also supports third-party network-based security mechanisms and secured communication channels.
- Replication Server uses optimistic concurrency control that processes failures when they occur. Compared to other methods, optimistic concurrency control provides greater data availability, uses fewer resources, and works well with heterogeneous data servers.

# Application Architecture for Replication Systems

The architectural design of your replication system determines the overall success of your replication environment. As an application designer and user, you need to explore the various options before you implement Replication Server.

## Application Types

Determining the type of Replication Server application you build determines, in part, the replication strategy you use.

Implementation Strategies discusses various replication scenarios.

Replication Server supports these basic application types:

- Decision support
- Distributed online transaction processing (OLTP)
- Remote OLTP using request functions
- Warm standby

Each of these application types differs in the way it updates primary data and in the way it distributes primary and replicated data within the replication system.

**See also**

- *Effects of Loose Consistency on Applications* on page 30
- *Methods for Updating Primary Data* on page 32
- *Implementation Strategies* on page 37

### Decision-Support Applications

Decision-support clients and production online transaction processing (OLTP) clients use data differently. Decision-support clients execute lengthy queries that hold locks on tables to guarantee serial consistency.

OLTP clients, on the other hand, execute transactions that must complete quickly and cannot accept the delays caused by decision-support clients' data locks. The two types of clients do not interfere with each other if they maintain separate copies of replicated tables.

Replication Server off-loads processing associated with decision-support applications from a centralized online transaction processing application onto local servers. The primary database manages the transaction processing, and the replicate databases at local sites handle requests for information from decision-support clients. Providing a separate, reference-only copy of the data allows OLTP systems to continue unobstructed.

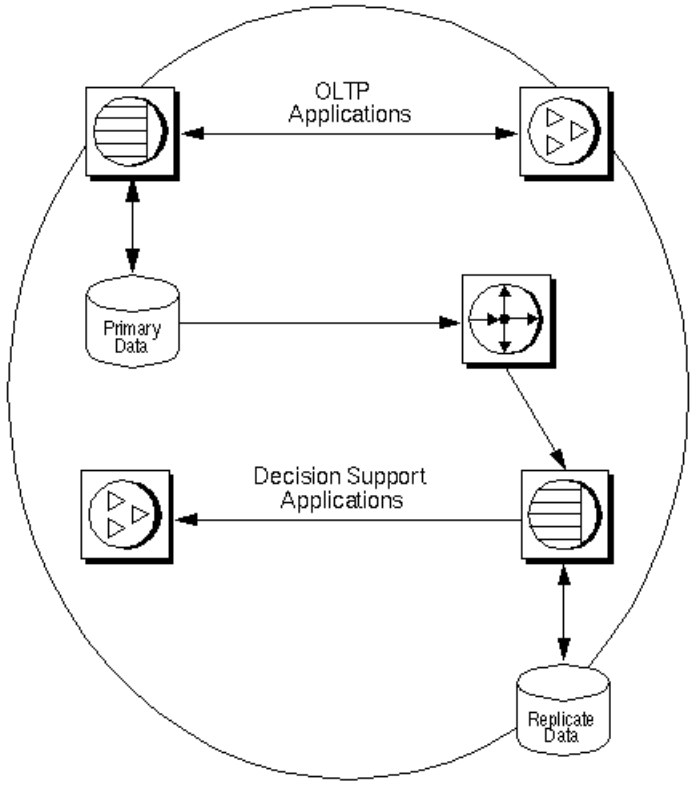
Multiple copies of tables containing primary data used in decision-support applications can be maintained at a single site or at multiple sites over the network.

### Multiple Copies at a Single Site

You can create subscriptions for multiple copies of a table by creating the table in different databases at the same site and then creating subscriptions for each one.

If OLTP and decision-support clients are on the same LAN, one Replication Server can manage both the primary data and the replicate data.

**Figure 5: Single LAN Decision-Support Replicate**




For best performance, the databases are usually maintained by different data servers. The subscriptions can request different subsets of the data to be maintained in each database, so the replicated copies do not have to be identical.

If you must have two copies of a table in the same database, you can use multiple replication definitions for a primary table. One replication definition could have `publishers` as the replicate table name, and the other `publishers2`. Multiple replication definitions are also useful if you want different replicates to receive different column subsets.

Another way to update multiple tables in an Adaptive Server database is to use stored procedures. Code the multiple updates in the stored procedures and write Replication Server function strings to execute the stored procedures. You can also use replicated functions and stored procedures to update multiple tables.

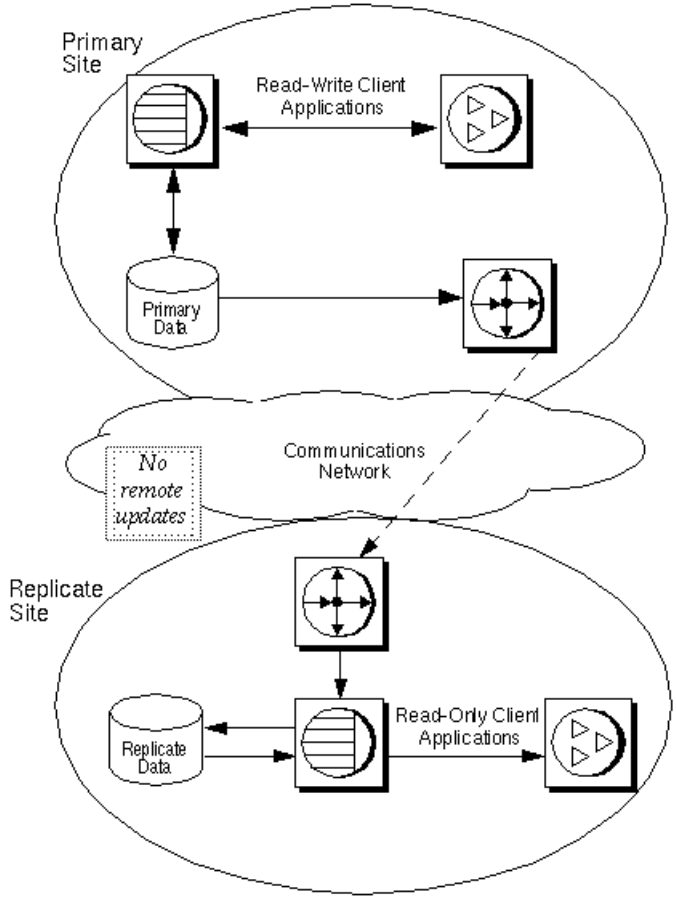
### See also

- *Multiple Replication Definitions* on page 57

## Multiple Copies Distributed over a Network

When copies of tables are distributed over a WAN in a decision-support application, all updates are performed by applications executing at the primary site and are distributed to the remote sites that have subscriptions for the data.

**Figure 6: Multiple LAN Decision-Support Replicate**



This type of system uses the centralized primary maintenance method of updating primary data. Clients at remote sites subscribe to replication definitions or publications of primary data. They do not update primary data.

### See also

- *Centralized Primary Maintenance* on page 32

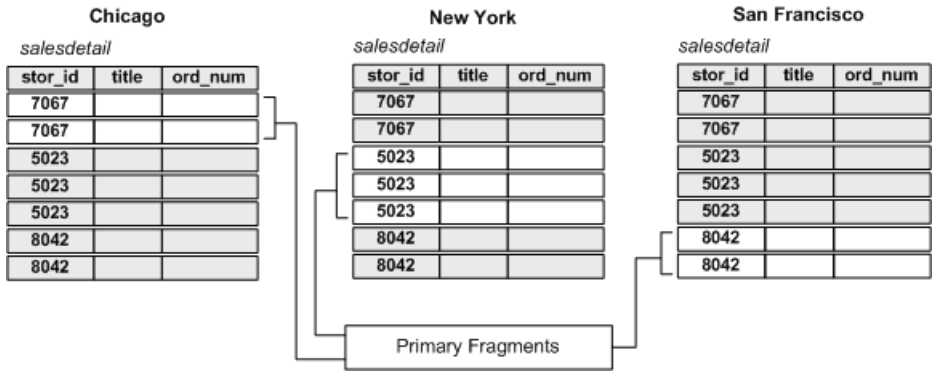
## Distributed OLTP Applications

Although some distributed transaction-processing applications maintain centralized primary data, others fragment primary data among replicate sites.

A primary fragment is a horizontal segment of a table that holds the primary version of a set of rows. Updates are first applied to the primary version and are then distributed to sites that have replicated copies of the data.

Sites that are responsible for, or own, portions of a table by definition have multiple primary fragments. For example, the `salesdetail` table has primary fragments in Chicago, New York, and San Francisco.

**Figure 7: Table with Multiple Primary Fragments**




A key constructed from one or more columns identifies the primary fragment where a row belongs. The key for the `salesdetail` table is the `stor_id` column.

- Rows with "7067" in the `stor_id` column belong to the primary fragment at the Chicago site.
- Rows with "5023" in the `stor_id` column belong to the primary fragment at the New York site.
- Rows with "8042" in the `stor_id` column belong to the primary fragment at the San Francisco site.

There are three application models based on multiple primary fragments:

- Distributed primary fragments – tables at each site contain both primary and replicated data. Updates to the primary version are distributed to other sites. Updates to nonprimary data are received from the primary site.
- Corporate rollup – multiple primary fragments maintained at remote sites are consolidated into a single aggregate replicate table at a central site.
- Redistributed corporate rollup – this model is the same as the corporate rollup model, except that the consolidated table is redistributed.

More information about these models can be found in Implemenation Strategies topic.

## Remote OLTP Using Request Functions

Use replicated functions to execute transactions remotely.

Client applications at remote sites can update primary data asynchronously with request functions. The client application does not require a network connection to the primary site, and the request can be accepted by the Replication Server even when the primary site is not available.

Once the request function executes the stored procedure in the primary database, Replication Server may replicate some or all of the data changes made in the primary database. These changes can be propagated to replicate databases as data rows or as applied functions.

### Local Update Applications

A local update application allows clients at a remote site to see the updates they have entered before the replication system returns them from the primary site.

For example, if a customer account is updated at a remote site, clients at the site can see the results of the transaction even if the primary site is not accessible.

Local updates can be performed by using a pending updates table. For each replicated table, a corresponding local table contains provisional updates—updates that have been submitted to the primary site, but that have not been returned through the replication system. Client applications update the pending transactions table and, at the same time, send a request function to the primary site.

When the update succeeds against the primary copy, it is distributed to remote sites, including the site where the transaction originated. You can create a function string or replicated stored procedure to update the replicated table and delete local updates from the pending table. This makes it possible for client applications to know which transactions have been confirmed and which are pending.

#### See also

- *An Example Using a Local Pending Table* on page 64

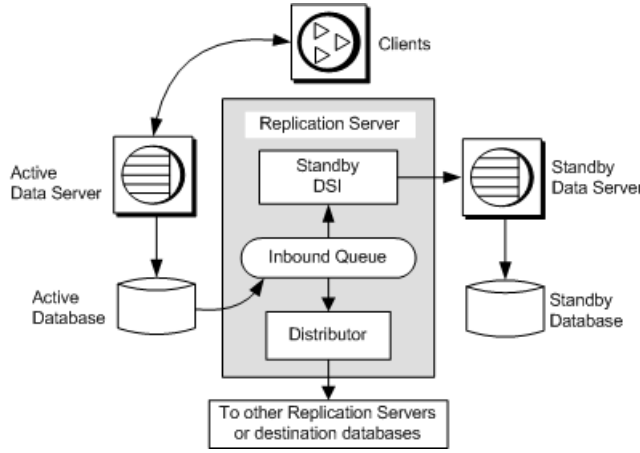
## Standby Applications

A warm standby application maintains a pair of databases from the same vendor, one of which functions as a standby copy of the other.

Client applications generally update the active database, while Replication Server maintains the standby database as a copy of the active database. Replication Server keeps the standby database consistent with the active database by replicating transactions retrieved from the active database transaction log.

If the active database fails, or if you need to perform maintenance on the active database or data server, you can switch to the standby database so that client applications can resume work with little interruption.

**Figure 8: Warm Standby System**




The two databases in a warm standby application appear as a single logical database in the replication system. Depending on your application, this logical database may not participate in replication, or it may be a primary database or a replicate database with respect to other databases in the replication system.

Several Replication Server and Replication Agent features explicitly support warm standby applications. See the *Replication Server Administration Guide Volume 2* for more detailed information about warm standby applications.

# Effects of Loose Consistency on Applications

Data in a replicate database is “loosely consistent” with data in the primary database.

Replicate data lags behind primary data by the amount of time it takes to distribute updates from the primary database to another part of the replication environment. This latency can be

measured in seconds (or less) when the system is working properly. If a component fails—if, for example, a network connection is temporarily lost—updates can be delayed for minutes, hours, or days. Thus, latency information can be used to monitor the performance and health of the replication environment.

Although replicate data may lag behind primary data, it is transactionally consistent with the primary data. Replication Server delivers transactions to replicate databases in the order they are committed in the primary database. This ensures that the replicate data goes through the same series of states as the primary data.

The importance of loose consistency varies by application and even within an application. Some applications tolerate average system lag time and occasional, longer delays with no special provisions. Some require special handling when the lag time becomes too great, and some require special handling for certain types of transactions.

### See also

- *Application Types* on page 25
- *Methods for Updating Primary Data* on page 32

## Risks in High-Value Transactions

Minimize data replication risks by differentiating between high-value and low-value transactions.

The delay introduced by data replication adds risk to some business decisions. For example, a banking application that approves cash withdrawals uses the most current account information available to verify that a customer's balance is sufficient to cover the withdrawal. If withdrawals processed in a primary database have not reached the replicate database, an application using the replicate database risks approving a withdrawal that exceeds the funds available in the customer's account.

To limit risk, the banking application can distinguish between high-value transactions and low-value transactions. For example, it might approve a $100 withdrawal based on the account balance in the local replicate database, but it would log in to the primary database to check the account balance before approving a $1000 withdrawal.

## Lag Time

Measure lag time for a replicate site to limit risks for some transactions. A small lag time indicates that the primary and replicate data are nearly consistent. An extensive lag time indicates a greater potential difference between the primary and replicate data.

An application can use a measure of lag time to:

- Limit risk by restricting the transactions clients can execute as the lag time increases. For example, a banking application could include the lag time in its approval formula. It might allow withdrawals of up to $1000 based on the balance in the local replicated table when

the latency is less than a minute. However, if the lag time is more than a minute, the application logs in to the primary database to approve withdrawals of more than $500.

- Provide clients with a "performance meter" for data replication. Clients can use an estimate of lag time as an advisory. For example, a decision-support user, noting that the lag time is high, might wait for the local data to catch up with the primary data, and for the lag time to decrease, before running an analysis based on replicate data.

See *Replication Server Administration Guide Volume 2 > Performance Tuning* for information on measuring latency.

# Methods for Updating Primary Data

In a replication system, the primary copy of a data row is the definitive copy. An update committed in the primary database is authoritative and is distributed to all databases with subscriptions for the data.

Replication Server distributes transactions after they are committed in the primary database. Because changes made to replicate data are not distributed, make the data in replicate databases read-only for clients and route all client transactions to the primary database.

There are four ways to update primary data in a replication system based on Replication Server:

- Primary data maintenance is centralized at the primary site. Clients cannot update primary data from remote sites.
- Clients at remote sites update primary data through network connections.
- Clients at remote sites update primary data using request functions.
- Primary data maintenance is distributed at multiple primaries. Any resulting conflicts must be avoided or resolved.

### See also

- *Application Types* on page 25
- *Effects of Loose Consistency on Applications* on page 30

## Centralized Primary Maintenance

Centralized primary maintenance method is the simplest, and the most restrictive, for remote clients. Client applications at remote sites use replicate data for reference only.

This architecture can be used to create a copy of a production OLTP system that allows decision support applications to run separately from the OLTP system.

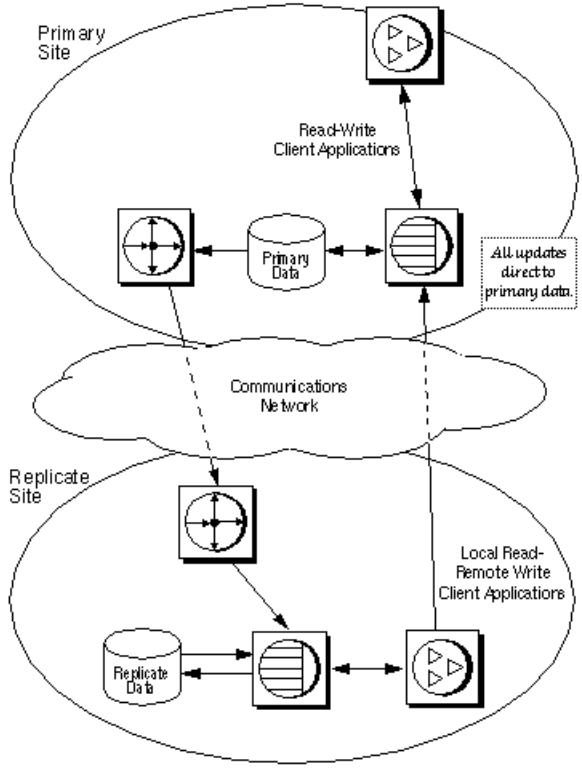
### See also

- *Decision-Support Applications* on page 25

## Primary Maintenance Via Network Connections

For some applications, clients at remote sites must update primary data. The easiest way to do this is for the client to connect directly to the primary data server through the network. Replication Server distributes updates from the primary to the remote sites in the usual way.

**Figure 9: Primary Data Maintenance via Network**




This architecture uses client connections through the WAN. It is useful for applications with large amounts of data and low update rates. Updates are executed directly against the primary data so that they can be distributed to all sites that have subscriptions for the data. At remote sites, local replicated copies of the data can be used for decision-support applications to reduce the network load.

## Managing Update Conflicts for Multiple Primaries

When there are multiple primaries, design remote updates so they do not introduce errors resulting from simultaneous requests to update the same information from multiple remote sites.

If updates from two different sites conflict when they are applied at a primary site, one of the updates will be rejected, and the replicated data at the site where the update was rejected will be inconsistent with primary data.

To handle intersite concurrency conflicts when there are multiple primaries:

- When each row has an owner – design the application so that intersite conflicts are impossible. For example, restrict the updates performed at one site to rows that cannot be updated by clients at other sites. This guarantees that updates will not conflict at the primary site.
- When there is no segmentation of ownership – add version control information to function strings to allow conflicts to be detected and handled.

### Design Conflicts Out of an Application

Handle conflicting updates from different sites.

Construct the application and the environment so that conflicts are impossible. For example, an application with customer account information distributed to each of several branch offices could require that an account be updated only at the customer's home branch. This prevents two clients from updating the same account at the same time in different databases.

Another technique is to include a location key, such as a branch ID, in the primary key for replicated tables. If each site uses a unique location key for all of its transactions, inter-site conflicts cannot occur.

### Version-Controlled Updates

Version-controlled updates can help detect and resolve various data update conflicts.

You can use version-control updates to:

- Detect and resolve conflicting updates
- Resolve multiple primary conflicts when there is no single primary source
- Make multiple requests to a single primary

Updates are accepted or rejected based on a version column that changes each time the row is updated. The version column can be a number that increases with each update, a timestamp, or some other value from a set of unique values.

To update a row, an application must provide the current value of the version column at the primary site. Typically, the value is provided as a parameter to a replicated stored procedure. The stored procedure at the primary site checks the version parameter and takes appropriate action if it detects a conflict. If the application chooses to roll back the transaction, it is written to the exceptions log.

**Note:** Managing update conflicts using version control requires careful planning and design. When there are multiple primaries, it is usually simpler and more effective to establish owners for each row of a table.

# Implementation Strategies

There are various models and strategies for implementing your replication system. Replication Server includes procedures and sample scripts that you can adapt to your own application.

## Overview of Models and Strategies

There are various data replication models for replicating data.

Models include:

- Basic primary copy model – centralized primary data, distributed replicate data
- Distributed primary fragments model – both primary and replicate data distributed through the replication system
- Corporate rollup – distributed primary data, centralized replicate data
- Redistributed corporate rollup – same as corporate rollup, but updates redistributed to replicate databases
- Warm standby applications– two databases, one serving as backup for the other, which together as a logical unit may participate in replication

Other model variations and strategies you can use:

- Multiple replication definitions
- Publications
- Request functions
- Pending tables
- Master/detail relationships

The type of application you are building, the way you update primary data, and the way you manage potential update conflicts determine the model you use to implement your replication application.

For instance, you can use the basic primary copy model to implement either a decision-support application or a low-volume distributed OLTP system. You might implement a decision-support application using either the basic primary copy model or the redistributed corporate rollup model, depending on whether your primary data is centralized or fragmented. A distributed OLTP application might be implemented using the distributed primary fragment model, with or without corporate rollup, depending on additional decision-support needs.

### See also

- *Model Variations and Strategies* on page 57

# Basic Primary Copy Model

The basic primary copy model allows you to replicate data from a primary database to destination databases.

This model is well suited to decision-support applications, although low-volume transaction-processing applications can update primary data remotely, either directly over the WAN or through request functions (replicated stored procedures). Primary data that is updated from remote sites can then be replicated back to subscribing sites.

You can implement the basic primary copy model by using any or all of the following:

- Table replication definitions
- Applied functions
- Request functions

## Table Replication Definitions

Table replication definitions allow you to replicate data from a primary source as read-only copies.

You can create one or many replication definitions for a primary table although a particular replicate table can subscribe to only one of them. See Multiple Replication Definitions for an example using multiple replication definitions.

You also can collect replication definitions in a publication and subscribe to all of them at one time with a publication subscription. See Publications for an example using publications.

For each table you want to replicate according to the basic primary copy model, you need to:

- Set up routes and connections between Replication Servers.
- Create the table you want to replicate in the primary database.
- Create the table (or tables) to which you want to replicate in destination databases.
- Create indexes and grant appropriate permissions on the tables.

*At the Primary Site:*

- Mark the primary table for replication using the **sp_setreptable** system procedure.
- Create one (or more) replication definitions for the table at the primary Replication Server.

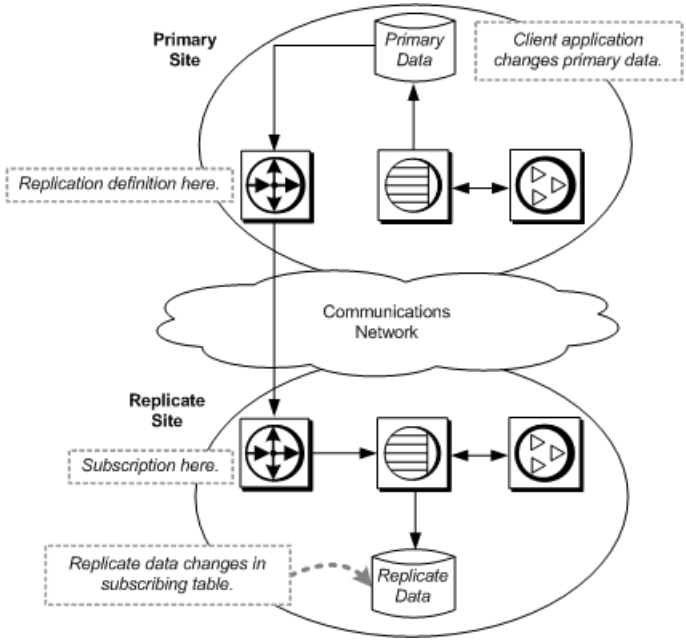
*At the Replicate Sites:*
Create subscriptions for the table replication definitions at each replicate Replication Server.

See the *Replication Server Administration Guide Volume 1* for details on setting up the basic primary copy model.

In this figure, a client application at the primary (Tokyo) site makes changes to the `publishers` table in the primary database. At the replicate (Sydney) site, the

`publishers` table subscribes to the primary `publishers` table—for those rows where `pub_id` is equal to or greater than 1000.

**Figure 10: Basic Primary Copy Model Using Table Replication Definitions**




*Marking the Table for Replication*

This script marks the `publishers` table for replication.

```
-- Execute this script at Tokyo data server
 -- Marks publishers for replication
 sp_setreptable publishers, 'true'
 go
 /* end of script */
```

*Replication Definition*

This script creates a table replication definition for the `publishers` table at the primary Replication Server.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition pubs_rep
create replication definition pubs_rep
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
 city varchar(20),
 state varchar(2))
primary key (pub_id)
```

```
go
/* end of script */
```

*Subscription*

This script creates a subscription for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
 -- Creates subscription pubs_sub
 Create subscription pubs_sub
 for pubs_rep
 with replicate at SYDNEY_DS.pubs2
 where pub_id >= 1000
 go
 /* end of script */
```

**See also**

- *Publications* on page 59
- *Multiple Replication Definitions* on page 57

## Applied Functions

You can use applied functions to replicate stored procedure invocations to remote sites with replicate data.

Using applied functions to replicate primary data, lets you:

- Reduce network traffic over the WAN
- Increase throughput and decrease latency because applied functions execute more rapidly
- Enable a more modular system design

In the following example, a client application at the primary (Tokyo) site executes a user stored procedure, **upd_publishers_pubs2**, which makes changes to the `publishers` table in the primary database. Execution of **upd_publishers_pubs2** invokes function replication, which causes the corresponding stored procedure, also named **upd_publishers_pubs2**, to execute on the replicate data server.

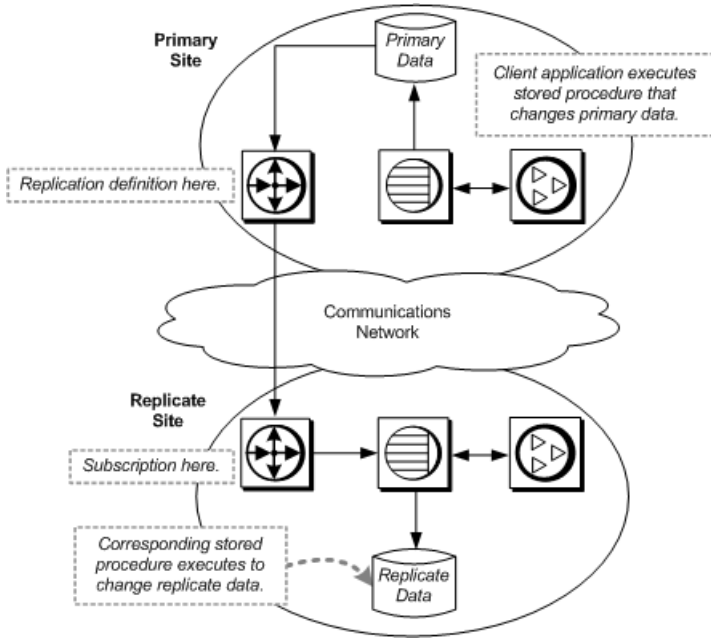
*At the Primary Site:*

- Create the user stored procedure in the primary database.
- Mark the user stored procedure for replicated function delivery using **sp_setrepproc.**
- Grant the appropriate procedure permissions to the appropriate user.
- At the primary Replication Server, create the function replication definition for the stored procedure with parameters and datatypes that match those of the stored procedure. You can specify only the parameters you want to replicate.

*At the Replicate Site:*

- Create a stored procedure in the replicate database with the same parameters (or a subset of those parameters) and datatypes as those created in the primary database. Grant appropriate permissions to the procedure to the maintenance user.
- Create a subscription to the function replication definition in the replicate Replication Server.

**Figure 11: Basic Primary Copy Model Using Applied Functions**




*Stored Procedures*

This script creates stored procedures for the `publishers` table at the primary and replicate sites.

```
-- Execute this script at Tokyo and Sydney data servers
 -- Creates stored procedure upd_publishers_pubs2
 create procedure upd_publishers_pubs2
(@pub_id char(4),
@pub_name varchar(40),
 @city varchar(20),
 @state char(2))
 as
 update publishers
 set
    pub_name = @pub_name,
    city = @city,
    state = @state
```

```
 where
    pub_id = @pub_id
 go
 /* end of script */
```

### *Function Replication Definition*

This script creates an applied function replication definition for the `publishers` table at the primary Replication Server. The replication definition uses the same parameters and datatypes as the stored procedure in the primary database.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
_upd_publishers_pubs2_repdef
   create applied function replication definition
   upd_publishers_pubs2_repdef
with primary at TOKYO_DS.pubs2
with all functions named upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
go
/* end of script */
```

### *Subscriptions*

You can create a subscription for a function replication definition in one of two ways:

- Use the **create subscription** command and the no-materialization method.

  Use this method if primary data is already loaded at the replicate, and updates are not in progress.
- Use the **define subscription**, **activate subscription**, and **validate subscription** commands and the bulk materialization method.

Using the No-Materialization Method

This script creates a subscription at the replicate Replication Server using the no-materialization method for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription using no-materialization
-- for upd_publishers_pubs2_repdef
create subscription upd_publishers_pubs2_sub
   for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
without materialization
go
/* end of script */
```

Using Bulk Materialization

This script defines, activates, and validates a subscription at the replicate Replication Server for the replication definition defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription using bulk materialization
-- for upd_publishers_pubs2_repdef
define subscription upd_publishers_pubs2_sub
   for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go
```

```
activate subscription upd_publishers_pubs2_sub
   for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go
/* Load data. If updates are in progress,use activate
subscription with the "with suspension" clause and
resume connection after the load. */
```

```
validate subscription upd_publishers_pubs2_sub
   for upd_publishers_pubs2_repdef
with replicate at SYDNEY_DS.pubs2
go
/* end of script */
```

## Distributed Primary Fragments Model

In a distributed primary fragments model, tables at each site contain both primary and replicated data.

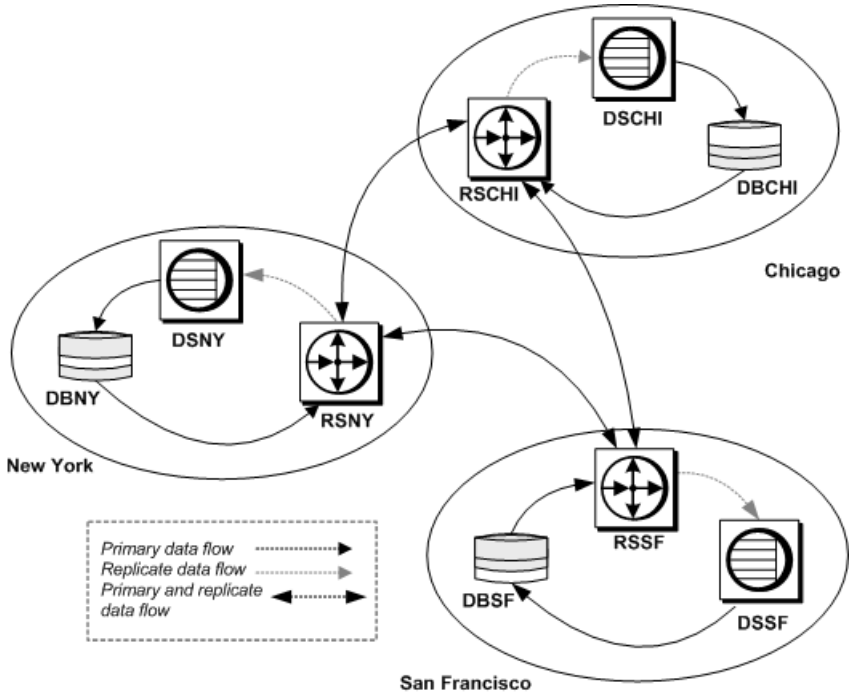
Each site functions as a primary site for a particular subset of rows called a fragment. Updates to the primary fragment are distributed to other sites. Updates to nonprimary data are received from the primary sites of other fragments.

Applications that use the distributed primary fragments model have distributed tables that contain primary and replicated data. The Replication Server at each site distributes modifications made to local primary data to other sites and applies modifications received from other sites to the data that is replicated locally.

To replicate a table in the distributed primary fragments model, you must perform these tasks at each site:

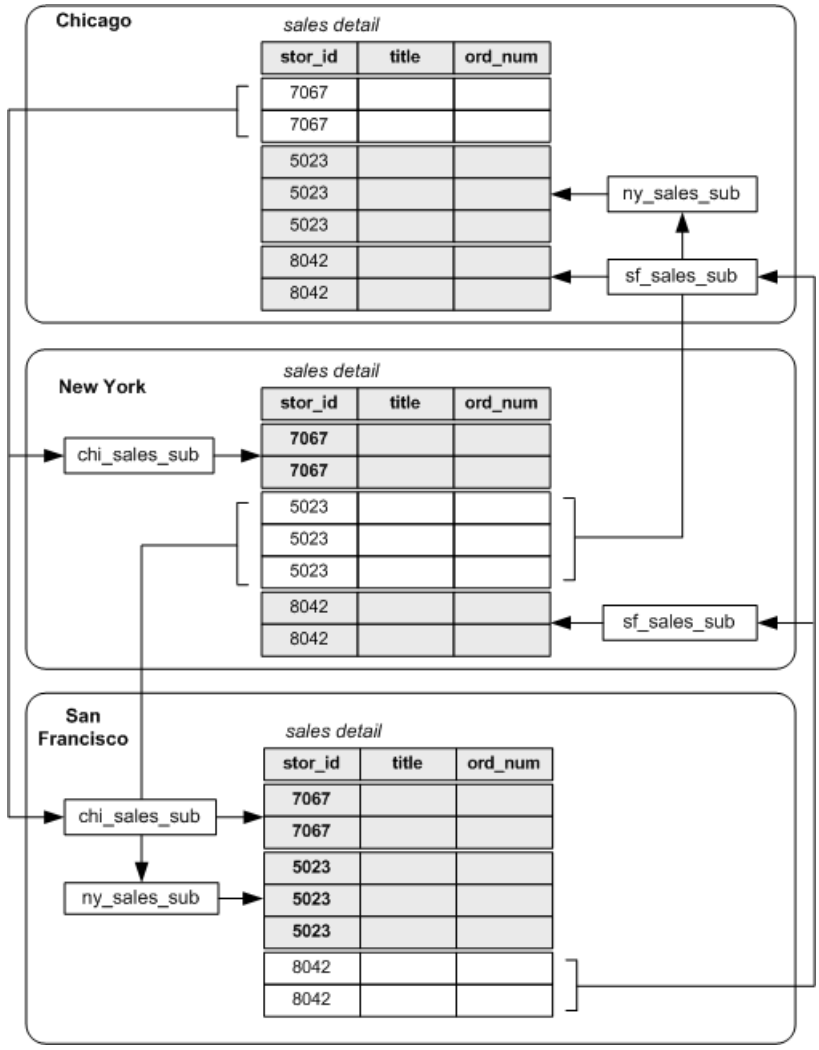
- Create a table in each database. The table should have the same structure in each database.
- Create indexes and grant appropriate permissions on the tables.
- Allow for replication on the tables using the **sp_setreptable** system procedure.
- Create a replication definition for the table at each site.
- At each site, create a subscription for the replication definition at the other sites. If *n* is the number of sites, create *n*-1 subscriptions.

**Figure 12: Distributed Primary Fragments Model**




This figure illustrates a `salesdetail` able set up with distributed primary fragments at three sites. Each site receives replicated data via two subscriptions.

**Figure 13: Table with Three Distributed Primary Fragments**




## Replication Definitions

Create replication definitions for the `salesdetail` table at each site using the sample scripts.

```
-- Execute this script at Chicago RSCHI.
-- Creates replication definition chi_sales.
create replication definition chi_sales_rep
with primary at DSCHI.DBCHI
with all tables named 'salesdetail'
```

```
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

```
-- Execute this script at New York RSNY.
-- Creates replication definition ny_sales.
create replication definition ny_sales_rep
with primary at DSNY.DBNY
with all tables named 'salesdetail'
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

```
-- Execute this script at San Francisco RSSF.
-- Creates replication definition sf_sales.
create replication definition sf_sales_rep
with primary at DSSF.DBSF
with all tables named 'salesdetail'
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
primary key (stor_id, ord_num)
searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

## Subscriptions

Each site has a subscription to the replication definitions at the other two sites.

These scripts create the subscriptions:

```
-- Execute this script at Chicago RSCHI.
-- Creates subscriptions to ny_sales and sf_sales.
create subscription ny_sales_sub
   for ny_sales_rep
   with replicate at DSCHI.DBCHI
   where stor_id = '5023'
go
create subscription sf_sales_sub
   for sf_sales_rep
   with replicate at DSCHI.DBCHI
   where stor_id = '8042'
```

```
go
/* end of script */
```

```
-- Execute this script at New York RSNY.
-- Create subscriptions to chi_sales and sf_sales.
create subscription chi_sales_sub
   for chi_sales_sub
   with replicate at DSNY.DBNY
   where stor_id = '7067'
go
create subscription sf_sales_sub
   for sf_sales_rep
   with replicate at DSNY.DBNY
   where stor_id = '8042'
go
/* end of script */
```

```
-- Execute this script at San Francisco RSSF.
-- Creates subscriptions to chi_sales and ny_sales.
create subscription chi_sales_sub
   for chi_sales_rep
   with replicate at DSSF.DBSF
   where stor_id = '7067'
go
create subscription ny_sales_sub
   for ny_sales_rep
   with replicate at DSSF.DBSF
   where stor_id = '5023'
go
/* end of script */
```

## Corporate Rollup

In the corporate rollup model, multiple primary fragments maintained at remote sites are consolidated into a single aggregate replicate table at a central site.

The corporate rollup model has distributed primary fragments and a single, centralized consolidated replicate table. The table at each primary site contains only the data that is primary at that site. No data is replicated to these sites. The corporate rollup table consists of rolled-up data from the primary sites.

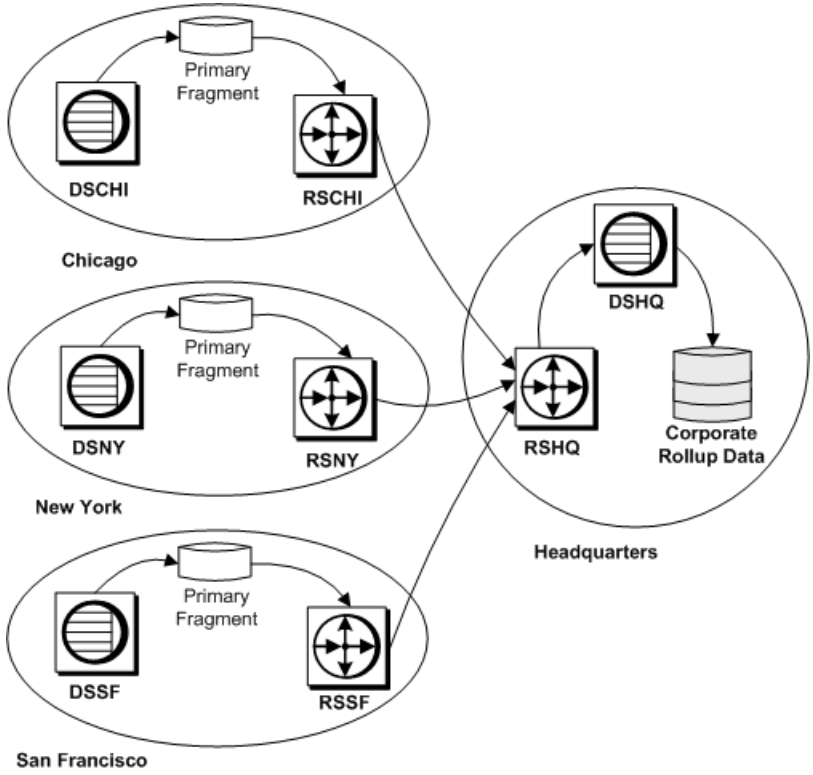
The corporate rollup model requires distinct replication definitions at each of the primary sites. The site where the data is consolidated has a subscription for the replication definition at each primary site.

Replication Agents are required at the primary sites but not at the central site, since data will not be replicated from that site.

These tasks must be performed to create a corporate rollup from distributed primary fragments:

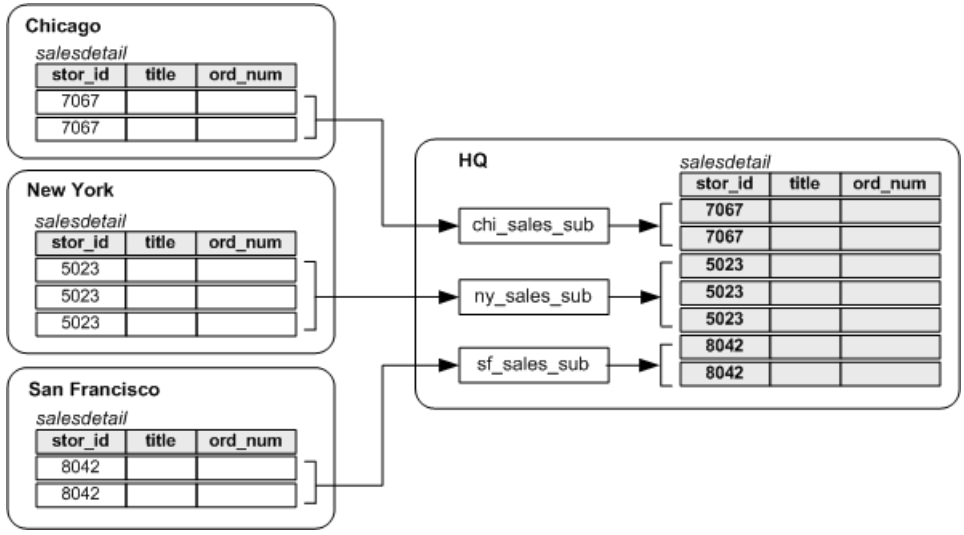
- Create the table in each primary database and in the database at the central site. The tables should have the same structure and the same name.
- Create indexes and grant appropriate permissions on the tables.
- In each remote database, allow for replication on the table with the **sp_setreptable** system procedure.
- Create a replication definition for the table at each remote site.
- At the headquarters site, where the data is to be consolidated, create subscriptions for the replication definitions at the remote sites.

**Figure 14: Corporate Rollup Model with Distributed Primary Fragments**




This figure illustrates a `salesdetail` table with a corporate rollup at a headquarters site. The headquarters site receives data from the remote sites via three subscriptions.

**Figure 15: Table with Multiple Primary Fragments**




## Replication Definitions

Create replication definitions for the `salesdetail` table at each primary site using the sample scripts.

```
-- Execute this script at Chicago RSCHI.
-- Creates replication definition chi_sales.
create replication definition chi_sales_rep
   with primary at DSCHI.DBCHI
   with all tables named 'salesdetail'
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
   primary key (stor_id, ord_num)
   searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

```
-- Execute this script at New York RSNY.
-- Creates replication definition ny_sales.
create replication definition ny_sales_rep
   with primary at DSNY.DBNY
   with all tables named 'salesdetail'
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
   primary key (stor_id, ord_num)
```

```
   searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

```
-- Execute this script at San Francisco RSSF.
-- Creates replication definition sf_sales.
create replication definition sf_sales_rep
   with primary at DSSF.DBSF
   with all tables named 'salesdetail'
   (stor_id char(4),
    ord_num varchar(20),
    title_id varchar6),
    qty smallint,
    discount float)
   primary key (stor_id, ord_num)
   searchable columns(stor_id, ord_num, title_id)
go
/* end of script */
```

## Subscriptions

The headquarters site has subscriptions to the replication definitions at each of the three primary sites. The primary sites have no subscriptions.

This script creates the subscriptions in the RSHQ Replication Server:

```
-- Execute this script at Headquarters RSHQ.
-- Creates subscriptions to chi_sales, ny_sales,
-- and sf_sales.
create subscription chi_sales_sub
   for chi_sales_rep
   with replicate at DSHQ.DBHQ
   where stor_id = '7067'
go
create subscription ny_sales_sub
   for ny_sales_rep
   with replicate at DSHQ.DBHQ
   where stor_id = '5023'
go
create subscription sf_sales_sub
   for sf_sales_rep
   with replicate at DSHQ.DBHQ
   where stor_id = '8042'
go
/* end of script */
```

## Redistributed Corporate Rollup

The redistributed corporate rollup model is the same as the corporate rollup model, except the consolidated table is redistributed back to the remote sites.

Primary fragments distributed at remote sites are rolled up into a consolidated table at a central site. At the site where the fragments are consolidated, RepAgent processes the consolidated table as if it were primary data.

The consolidated table is described with a replication definition. Other sites can create subscriptions for this table.

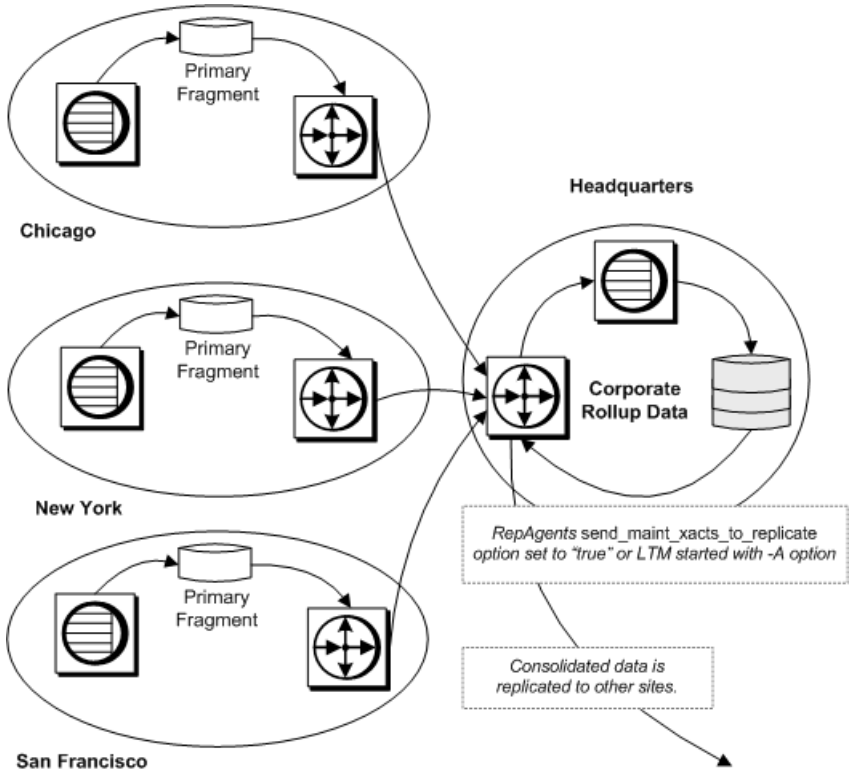
Normally, RepAgent for Adaptive Server filters out updates made by the maintenance user, ensuring that replicated data is not redistributed as primary data.

The RepAgent **send_maint_xacts_to_replicate** option is provided for the redistributed corporate rollup model. If you start RepAgent with **send_maint_xacts_to_replicate** set to "true," RepAgent submits all updates to the Replication Server as if they were made by a client application.

If you use the redistributed corporate rollup model, do not allow:

- Primary sites to resubscribe to their primary data. If they do, transactions could loop endlessly through the system.
- Applications to update the corporate rollup table. All updates should originate from the primary sites.

**Figure 16: Redistributed Corporate Rollup with Distributed Fragments**




The design of the redistributed corporate rollup model is identical to the corporate rollup model, except that:

- RepAgent must be installed at the headquarters site for the DBHQ database. RepAgent must be started with the **send_maint_xacts_to_replicate** option set so that it will transfer log records from the maintenance user.
- RepAgent is required for the RSHQ RSSD, since data will be distributed from that site.
- A replication definition must be created for the `salesdetail` table at the headquarters site. Other sites can create subscriptions to this replication definition, but the primary sites must not subscribe to their own primary data.
- The RSHQ Replication Server must have routes to the other sites that create subscriptions for the consolidated replicate table. If the primary sites create subscriptions, routes must be created to them from RSHQ.

## Warm Standby Applications

In a warm standby application, Replication Server maintains a pair of databases from the same vendor, one of which acts as the backup of the other.

Typically, client applications update the active database while Replication Server maintains the other database as a standby copy of the active database. If the active database fails, or if you need to perform maintenance on the active data server or database, you can switch to the standby database (and back) with little interruption of client applications.

In a warm standby application, create:

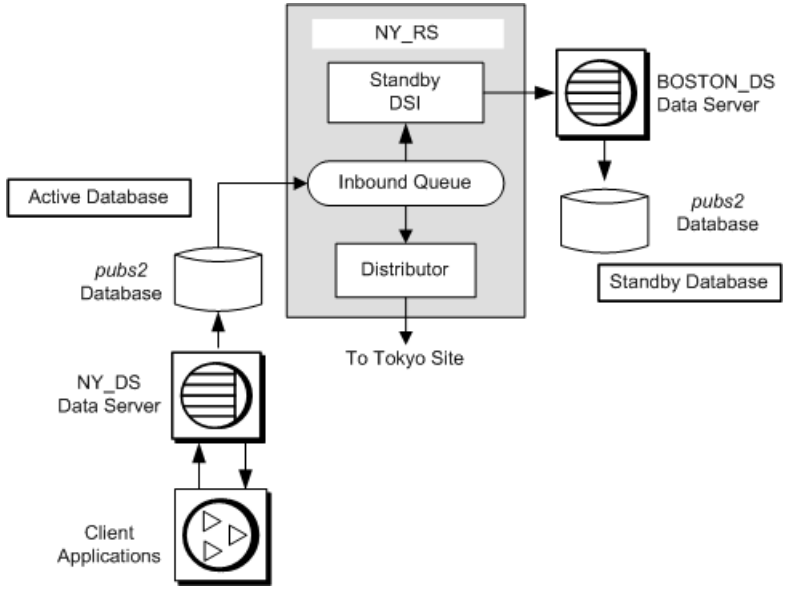
- A logical connection that Replication Server maps to the currently active database
- A physical connection for the active database
- A physical connection for the standby database

The logical database in a warm standby application may, with respect to other databases in the replication system, function as one of:

- A database that does not participate in replication
- A primary database
- A replicate database

This figure illustrates a warm standby application operating on the BOSTON_DS data server for a `pubs2` database on the NY_DS data server. The database is replicated to TOKYO_DS.

**Figure 17: Warm Standby System**



In this scenario, the `pubs2` database acts as the primary database. The primary `pubs2`database for which a standby is created is called the active database.

## Setting Up a Warm Standby Application

Learn how to set up a warm standby application for an active database, using an example wherein an active Adaptive Server database is already established.

Review the warm standby information in the *Replication Server Administration Guide Volume 1* before proceeding. For setting up a warm standby application for Oracle, see *Heterogeneous Replication Guide*.

1. Use **sp_reptostandby** to mark the entire active database for replication to the standby database.

   **sp_reptostandby** enables replication of data manipulation language (DML) and supported data definition language (DDL) commands and stored procedures. See the *Replication Server Administration Guide Volume 2 > Managing Warm Standby Applications* for detailed information.
2. Reconfigure RepAgent using the **sp_config_rep_agent** stored procedure with the **send_warm_standby_xacts** option. Restart RepAgent.
3. Grant **replication_role** to the active database maintenance user.
4. On the active data server, add the maintenance user of the standby database to the active database, and grant **replication_role** to the new maintenance user. This step ensures that the maintenance user ID exists in the standby database after the database is loaded (step 8).
5. Log in to the Replication Server that is to manage the warm standby database, and create a logical connection for the active database, using the **create logical connection** command. The name of the logical connection must be the same as the name of the active database.

   **Note:** If you create the logical connection before you create the active database connection, use different names for the logical connection and the active database.
6. On the standby data server, create the standby database with the same size as the active database.
7. Use Sybase Central or **rs_init** to create the standby database connection. For more information, see the Replication Server online help and the Replication Server installation and configuration guide for your platform.

   After the connection is created, log in to Replication Server and use the **admin logical_status** command to make sure that the new connection is “active.”
8. Initialize the standby database using **dump** and **load** without the **rs_init** “dump marker” option. (Or you can use **bcp**. Refer to the *Replication Server Administration Guide Volume 2* for more information.)

   a) On the Replication Server, suspend the active database connection.

**Note:** If you cannot suspend the active database, use **dump** and **load** with the **rs_init** "dump marker" option.

b) On the active Adaptive Server, dump the active database.
c) Load the active database dump into the standby database.
d) On the standby Adaptive Server, put the standby database online.

**9.** On the Replication Server, resume connections to the active and standby databases, using the **resume connection** command.

Check the logical status, using the **admin logical_status** command. Do not continue unless both active and standby databases are marked "active."

**10.** Verify that modifications occur from active to standby database.

Using **isql**, update a record in the active database and then verify the update in the standby database.

## Switching to the Standby Database

Switch between the active and standby databases.

When you switch from the active database to the standby database, prevent client applications from executing transactions against or updating the active database during the switch. When the switch completes, clients can connect to the new active database to continue their work.

Determine whether a switch is necessary:

- Do not switch if the active data server is experiencing a transient failure, which is a failure from which the Adaptive Server recovers when restarted, without additional recovery steps.
- Do switch if the active database will be unavailable for a long period of time.

You must use the **switch active** command to switch the active and standby databases.

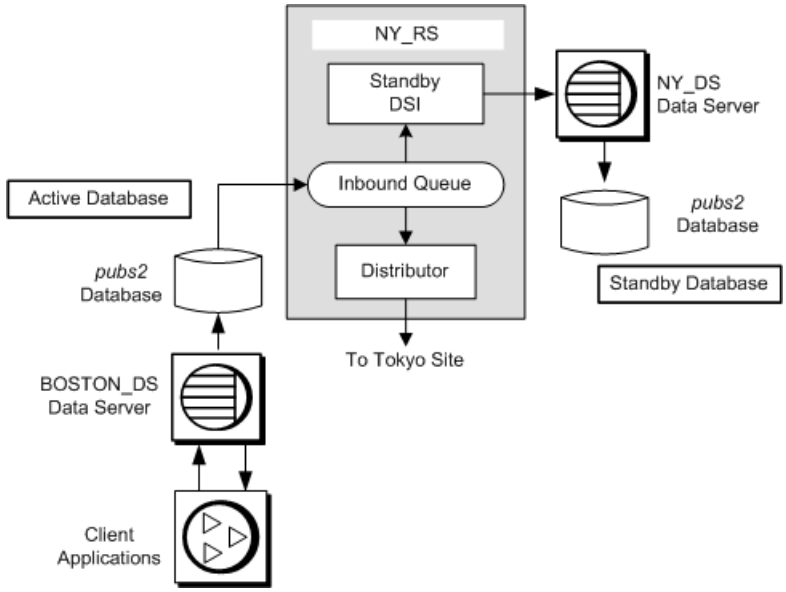
**1.** On the Replication Server, use **switch active** to switch processing to the standby database.
**2.** Monitor progress of the switch. The switch is complete when the standby connection is active and the previously active connection is suspended.
   a) On the Replication Server, check the logical status, using the **admin_logical_status** command.
   b) To follow the progress of the switch, check the last several entries in the Replication Server error log.
**3.** Start the RepAgent for the new active database.
**4.** Decide what you want to do with the old active database. You can:

   - Bring the database online as the new standby database, and resume connections so that Replication Server can apply new transactions,
   - Drop the database connection using the **drop connection** command. You can add it again later as the new standby database.

- Create a new warm standby database connection for the new active database if the new standby database is not the original active database.

5. Using **isql**, update a record in the new active database, and then check the update the new standby database.

**Figure 18: Warm Standby System, After Switching**




## Switch Clients to the New Database

Switching from the active to the standby database does not switch client applications to the new active data server and database.

You must devise a method to handle client switching. For example, you could:

- Set up two interfaces files, one for client applications and one for Replication Server. At switch time, modify the client interfaces file to point to the new active server.
- Create an interfaces file entry with a symbolic data server name for use by client applications. At switch time, modify the address information associated with the symbolic name.
- Use a mechanism, such as an intermediate Open Server, to map the client application data server connections to the currently active data server automatically.

See the *Replication Server Administration Guide Volume 2*.

# Model Variations and Strategies

Model variations and other strategies are useful when implementing your replication system design.

Some of the model variations and strategies are:

- Multiple replication definitions – a strategy using multiple replication definitions for a primary table to specify different table names, column sets, and column names, thereby presenting differing views of the primary table to subscribing replicate tables.
- Publications – a strategy that allows users to subscribe to a set of replication definitions with a single subscription.
- Pending tables – a strategy used with request functions that allows users to see the results of updates to primary data before the update has been returned to the replicate site.
- Implementing master/detail relationships – uses request functions and stored procedures to ensure proper subscription migration.

## Multiple Replication Definitions

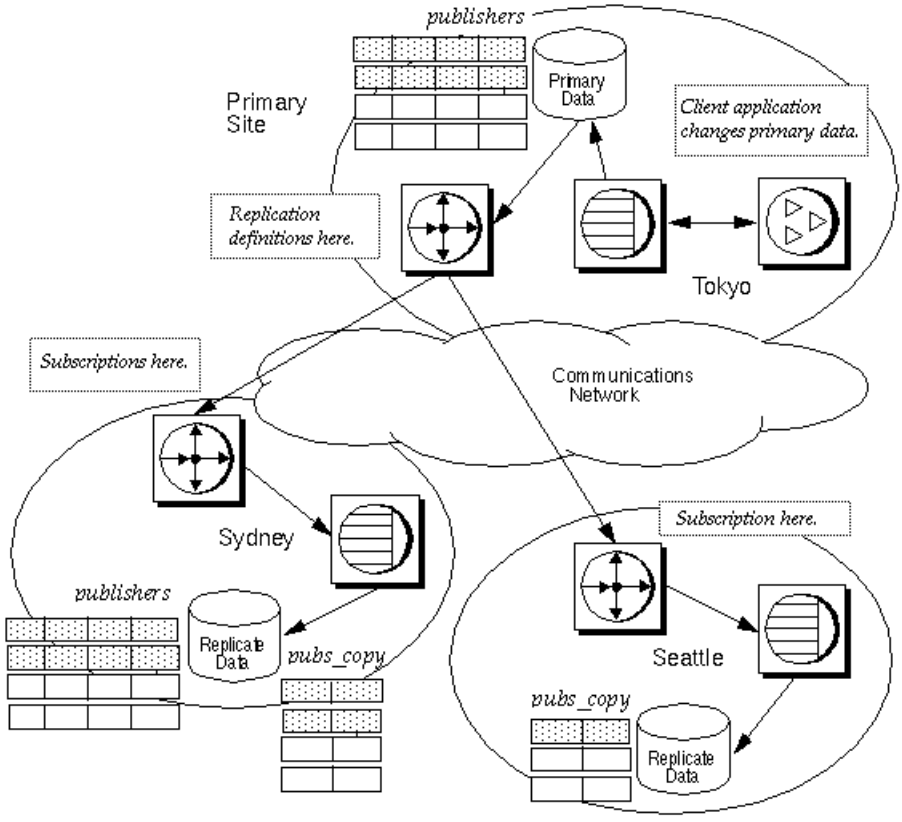
You can create multiple replication definitions for a single primary table. Each replication definition can specify different table names, column sets, and column names, thereby presenting different views of the primary table to each of the subscribing replicate tables.

**Note:** You can create multiple replication definitions for a primary table, and a replicate table can subscribe to multiple table replication definitions. However, a replicate table can subscribe only to one replication definition per primary table.

To set up a system using multiple replication definitions, follow the directions in Table Replication Definitions, creating multiple replication definitions as needed and a subscription for each one. When you use multiple replication definitions, you are using a variation of the primary copy model.

In this figure, a client application at the primary (Tokyo) site makes changes to the `publishers` table in the primary database. At one replicate (Sydney) site, the `publishers` table subscribes to the complete table, and the `pubs_copy` table subscribes only to the `pub_id` and `pub_name` columns. At another replicate (Seattle) site, the `pubs_copy` table subscribes to the `pub_id` and `pub_name` columns, where `pub_id` is equal to or greater than 1000.

**Figure 19: Multiple Replication Definitions**




**See also**

- *Table Replication Definitions* on page 38

## Replication Definitions

Create table replication definitions for the `publishers` table at the primary Replication Server using the sample scripts.

Each replication definition describes a different view of the primary table.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definitions pubs_rep and
-- pubs_copy_rep
create replication definition pubs_rep
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
```

```
 city varchar(20),
 state varchar(2))
primary key (pub_id)
go

create replication definition pubs_copy_rep
with primary at TOKYO_DS.pubs2
with primary table named 'publishers'
with replicate table named 'pubs_copy'
(pub_id char(4),
 pub_name varchar(40))
primary key (pub_id)
go
/* end of script */
```

### Subscriptions

Create subscriptions for the replication definitions defined at the primary Replication Server.

These scripts create subscriptions for the replication definitions defined at the primary Replication Server.

```
-- Execute this script at Sydney Replication Server
 -- Creates subscription pubs_sub and pubs_copy_rep
 Create subscription pubs_sub
 for pubs_rep
 with replicate at SYDNEY_DS.pubs2
 go

 create subscription pubs_copy_sub
 for pubs_copy_rep
 with replicate at SYDNEY_DS.pubs2
 go
 /* end of script */
```

```
-- Execute this script at Seattle Replication Server
 -- Creates subscription pubs_copy_sub
 create subscription pubs_copy_sub
 for publ_copy_rep
 with replicate at SEATTLE_DS.pubs2
 where pub_id >= 1000
 go
 /* end of script */
```

## Publications

Use publications to collect replication definitions for tables and/or stored procedures and then subscribe to them as a group. Publication usage is not a separate model; it provides a grouping technique that can be used by any model. With publications, you can monitor the status of one publication subscription for a set of tables and procedures.

When you use publications, you create and manage the following objects:

- Articles – replication definition extensions for tables or stored procedures that let you put table or function replication definitions in a publication.

- Publications – groups of articles from the same primary database.
- Publication subscriptions – subscriptions to a publication. When you create a publication subscription, Replication Server creates a subscription for each of the publication's articles.

The following steps summarize the procedure for replicating data using publications.

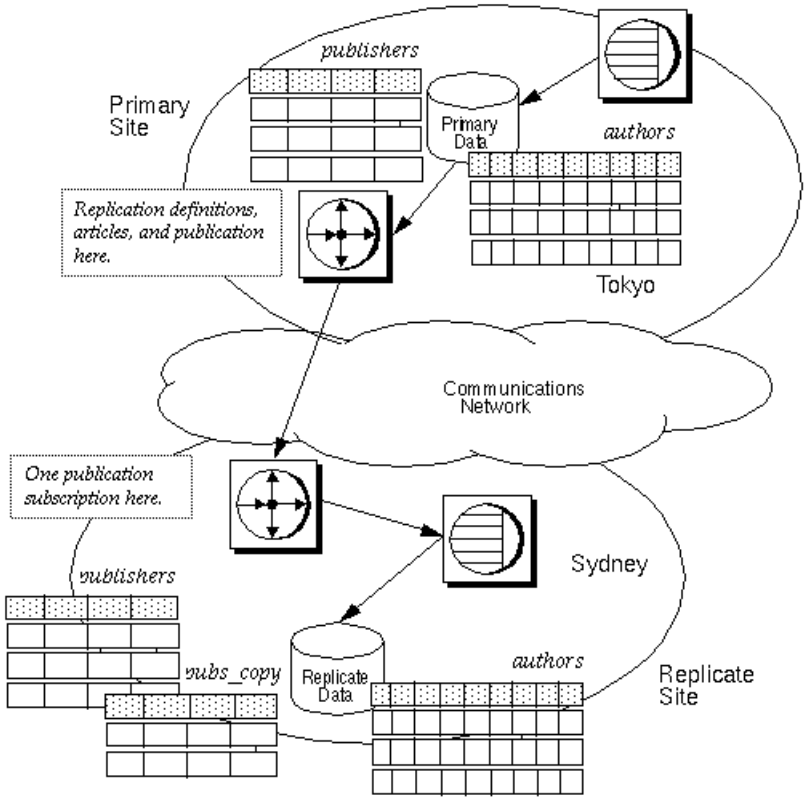
*At the Primary Site:*

1. Create or select the replication definitions to include in the publication.
2. Create the publication, using the **create publication** command.
3. Create articles that reference the replication definitions you have chosen, using the **create article** command.
4. Validate the publication using the **validate publication** command.

*At the Replicate Site:*
Create a subscription for the publication using the **create subscription** command.

The figure illustrates a table replication definition `pubs_rep`, referenced by two articles, and a function replication definition, referenced by one article, are collected in the publication `pubs2_pub`.

**Figure 20: Publications**




## Stored Procedure

Creates a stored procedure **update_ authors_pubs2** that updates the `authors` table in the `pubs2` database using the sample scripts.

Create the same procedure at the primary and replicate sites:

```
-- Execute this script at the Tokyo and Sydney
-- data servers
-- Creates the stored procedure update_authors_pubs2
create procedure upd_authors_pubs2
(@au_id id,
 au_lname varchar(40),
 au_fname varchar(20),
 phone char(12),
 address varchar(12),
 city varchar(20),
 state char(2),
 country varchar(12),
```

```
 postalcode char(10))
as
 update authors
 set
   au_lname = @varchar(40),
   au_fname = @varchar(20),
   phone = @char(12),
   address = @varchar(12),
   city = @varchar(20),
   state = @char(2),
   country = @varchar(12),
   postalcode = @char(10)
 where au_id = @au_id
go
/* end of script */
```

### Function Replication Definition

Create an applied function replication definition at the primary site using the sample script.

```
-- Execute this script at Tokyo Replication Server
-- Creates the applied function replication definition
-- upd_authors_rep_repdef
create applied function replication definition
upd_authors_rep
with primary at TOKYO_DS.pubs2
with all functions named upd_authors_pubs2
(@au_id id,
 au_lname varchar(40),
 au_fname varchar(20),
 phone char(12),
 address varchar(12),
 city varchar(20),
 state har(2),
 country varchar(12),
 postalcode char(10))
go
 /* end of script */
```

### Table Replication Definition

Create a table replication definition for the `publishers` table at the primary Replication Server using the sample script.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definitions pubs_rep
create replication definition pubs_rep
with primary at TOKYO_DS.pubs2
with all tables named 'publishers'
(pub_id char(4),
 pub_name varchar(40),
 city varchar(20),
 state varchar(2))
primary key (pub_id)
```

```
go
/* end of script */
```

### Publication

Create the publication `pubs2_pub` at the primary Replication Server using the sample script.

```
-- Execute this script at Tokyo Replication Server
 -- Creates publication pubs_pub
 create publication pubs2_pub
 with primary at TOKYO_DS.pubs2
 go
 /* end of script */
```

### Articles

Create articles for the publication `pubs2_pub` at the primary Replication Server using the sample script.

The sample script creates two articles for the replication definition, `pubs_rep`.

```
-- Execute this script at Tokyo Replication Server
-- Creates articles upd_authors_art, pubs_art, and
-- pubs_copy_art
create article upd_authors_art
 for pubs2_pub
with primary at TOKYO_DS.pubs2
with replication definition upd_authors_rep
go
```

```
create article pubs_art
 for pubs2_pub
with primary at TOKYO_DS.pubs2
with replication definition pubs_rep
go
```

```
create article pubs_copy_art
 for pubs2_pub
with primary at TOKYO_DS.pubs2
 with replication definition pubs_rep
 where pub_id >= 1000
 go
 /* end of script */
```

### Validation

Change the status of the publication `pubs2_pub` to “valid” using the sample script.

```
-- Execute this script at Tokyo Replication Server
-- Validates the publication pubs2_pub
validate publication pubs2_pub
with primary at TOKYO_DS.pubs2
go
/* end of script */
```

### Subscription

Create the subscription `pubs2_pub_sub` for the publication `pubs2_pub` using the sample script.

When this script is run, Replication Server creates article subscriptions for `upd_authors_art`, `pubs_art`, and `pubs_copy_art`.

```
-- Execute this script at Sydney Replication Server
-- Creates publication subscription pubs2_pub_sub
create subscription pubs2_pub_sub
 for publication pubs2_pub
with primary at TOKYO_DS.pubs2}
with replicate at SF.pubs2
without materialization
go
/* end of script */
```

## Request Functions

Request functions allow primary database users to invoke stored procedures on replicate data.

### An Example Using a Local Pending Table

The pending table allows clients at a remote site to update central data and see the updates at the remote site before they are returned from the central site. Use this model to implement local update applications.

In this strategy, a client application at a remote site executes a user stored procedure that updates data at the central site using a request function. Changes to the central data are replicated to the remote site via an applied function. A local pending table lets clients at the remote site see pending updates at the remote site before the replication system returns the updates.

When a client application executes the user stored procedure at the remote data server, it:

- Causes an associated stored procedure to execute and update data at the primary site
- Enters those updates in the local pending table

When the update succeeds at the central database, it is distributed to remote sites, including the site where the transaction originated. At the remote site, a stored procedure updates the replicated table and deletes the corresponding updates from the pending table.

To use applied functions, request functions, and a local pending table, you must complete these tasks.

*At the Remote Site:*

- Create a pending table in the remote database. Grant appropriate permissions.
- Create a user stored procedure in the remote database that initiates the request function and inserts data updates into the pending table.

- Mark the user stored procedure for replicated function delivery using **sp_setrepproc**.
- Grant procedure permissions to the appropriate user.
- Create a user stored procedure in the remote database that updates the remote table and deletes the corresponding update from the pending table. Grant appropriate permissions to the maintenance user.
- Create the request function replication definition for the request function.
- Create a subscription to the applied function replication definition created at the central site.

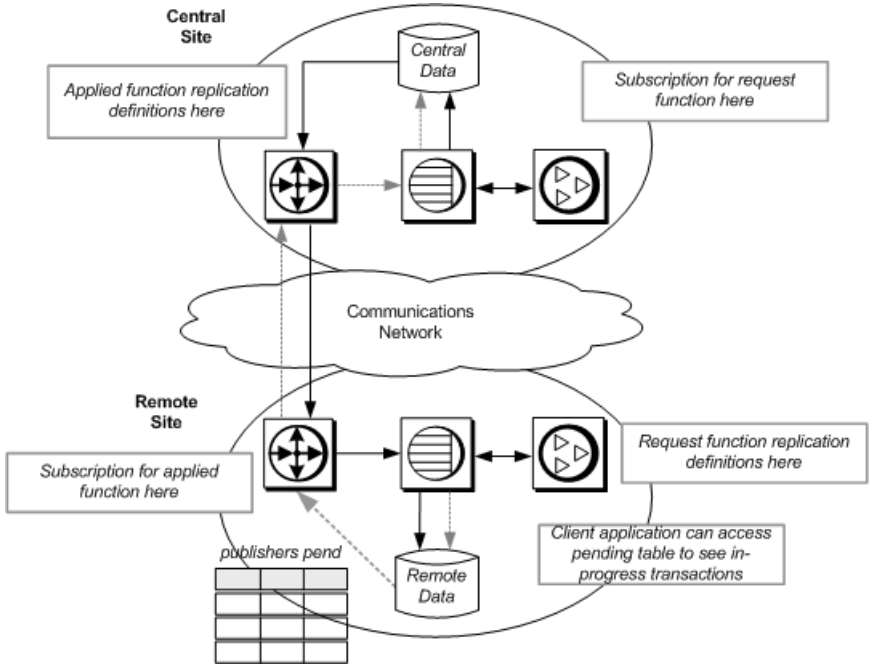
*At the Primary Site:*

- Create the stored procedure that modifies the central data.
- Create the applied function replication definition for the applied function.
- Create a subscription to the request function replication definition.

In this example, a client application at the remote (Sydney) site executes a stored procedure **upd_publishers_pubs2_req**, which inserts values in the `publishers_pend` table and causes an associated stored procedure, **upd_publishers_pubs2**, to execute at the central (Tokyo) site. Execution of **upd_publishers_pubs2** at the central site causes the stored procedure **upd_publishers_pubs** to execute at the remote site, which updates the `publishers` table and deletes the corresponding information from the `publishers_pend` table.

This figure illustrates the data flow when you use applied functions, request functions, and a local pending table. The gray arrows show the flow of the request function delivery. The black arrows show the flow of the applied function delivery.

**Figure 21: Request Functions and a Local Pending Table**




*Pending Table*

Create a pending table in the remote database using the sample script.

```
-- Execute this script at Sydney data server
-- Creates local pending table
create table publishers_pend
(pub_id char(4) not null,
 pub_name varchar(40) null,
 city varchar(20) null,
 statechar(2) null)
go
/* end of script */
```

*Stored Procedures*

Create the stored procedure **upd_publisher_pubs2** at the central (Tokyo) site using the sample scripts.

```
-- Execute this script at Tokoyo data server
-- Creates stored procedure
create procedure upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
as
   insert into publishers
```

```
   values (@pub_id, @pub_name, @city, @state)
go
/* end of script */
```

This script creates the **upd_publishers_pub2_req** stored procedure at the remote (Sydney) site. The **insert into** clause inserts values into the `publishers_pend` table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure
create procedure upd_publishers_pubs2_req
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
as
    insert into publishers_pend
    values (@pub_id, @pub_name, @city, @state)
go
/* end of script */
```

This script creates the **upd_publishers_pubs2** procedure for the remote (Sydney) site. It updates the `publishers` table and deletes the corresponding information from the `publishers_pend` table.

```
-- Execute this script at Sydney data server
-- Creates stored procedure upd_publishers_pubs2
 create procedure upd_publishers_pubs2
 (@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
 as
 update publishers
 set
    pub_name = @pub_name,
    city = @city,
    state = @state
 where
    pub_id = @pub_id
 delete from publishers_pend
 where
    pub_id = @pub_id
 go
 /* end of script */
```

### *Function Replication Definitions*

Create the applied function replication definition at the central (Tokyo) Replication Server using the sample scripts.

```
-- Execute this script at Tokyo Replication Server
-- Creates replication definition
create applied function replication definition
   upd_publishers_pubs2
with primary at TOKYO_DS.pubs2
(@pub_id char(4),
```

```
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
go
/* end of script */
```

This script creates the request function replication definition at the remote (Sydney) Replication Server:

```
-- Execute this script at Sydney Replication Server
-- Creates replication definition
create request function replication definition
   upd_publishers_pubs2_req
with primary at SYDNEY_DS.pubs2
with primary function named upd_publishers_pubs2_req
with replicate function named upd_publishers_pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2))
go
/* end of script */
```

*Subscription*

Create a subscription at the remote Replication Server using the no-materialization method for the applied function replication definition defined at the central Replication Server using the sample scripts.

```
-- Execute this script at Sydney Replication Server
-- Creates subscription using no-materialization
for upd_publishers_pubs2
create subscription upd_publishers_pubs2_sub
for upd_publishers_pubs2
with replicate at SYDNEY_DS.pubs2
without materialization
go
/* end of script */
```

This script creates a subscription at the central Replication Server using the no-materialization method for the request function replication definition defined at the remote Replication Server.

```
-- Execute this script at Tokoyo Replication Server
-- Creates subscription using no-materialization
for upd_publishers_pubs2_req
create subscription upd_publishers_pubs2_req_sub
for upd_publishers_pubs2_req
with replicate at TOKYO_DS.pubs2
without materialization
go
/* end of script */
```

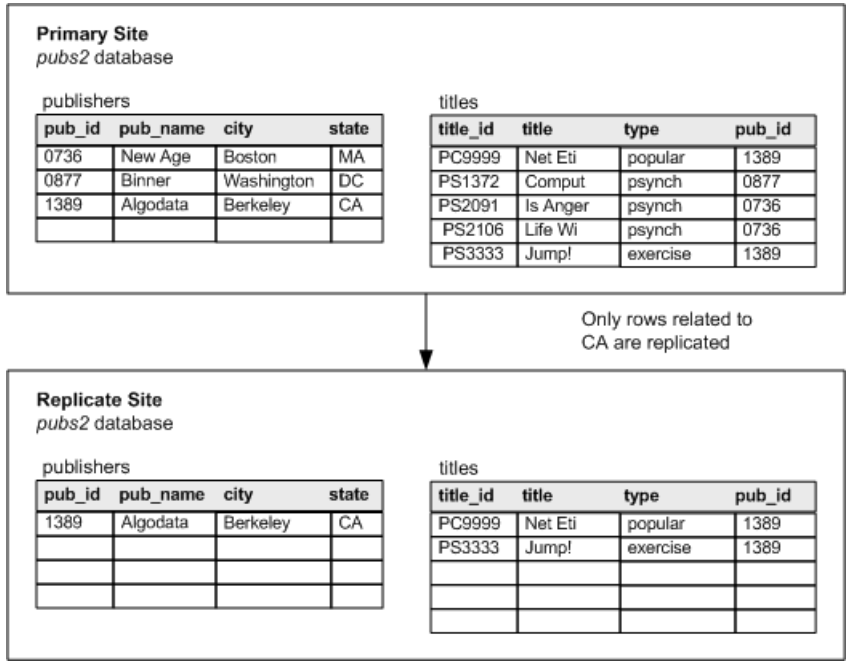
## Master/Detail Relationships Implementation

You can use applied functions to replicate only selected data to remote sites, thus reducing network traffic.

To implement master/detail relationships, use applied functions to support selective subscription to the master/detail tables. In this example,

- The `publishers` and `titles` tables exist in the `pubs2` database at the primary and replicate sites.
- NY_DS is the primary site data server and SF_DS is the replicate site data server.

**Figure 22: Sample Tables Used in Master/Detail Relationship**




The primary site contains all records, but the replicate site is interested only in records related to the state of California (CA). Only a selection of `publishers` and `titles` records need to be replicated, based on the `state` column. However, only the `publishers` table contains a `state` column.

Adding a `state` column to the `titles` table adds redundancy to the system. A second, more efficient solution ties updates to master and detail tables through stored procedures and then replicates the stored procedures using applied functions. The logic to maintain selective subscription is contained in the stored procedures.

For example, if, at the primary site, a publisher's state is changed from NY to CA, a record for that publisher must be inserted at the replicate site. Having replicate rows inserted or deleted as a result of updates that cause rows in a subscription to change is called subscription migration.

To ensure proper subscription migration, subscriptions are needed for a set of "upper-level" stored procedures that control the stored procedures that actually perform the updates. It is the invocation for the upper-level stored procedure that is replicated from the primary site to the replicate site.

To handle changes in the `state` column, the replicate site must subscribe to updates when either the new state or the old state is CA.

Perform these steps at the primary and the replicate sites to enable selective substitution at the replicate Replication Server.

*At the Primary and Replicate Sites:*

- Create stored procedures that insert records into the `publishers` and `titles` tables and an upper-level stored procedure that controls the execution of both insert procedures.
- Create stored procedures that delete records from the `publishers` and `titles` tables and an upper-level stored procedure that controls the execution of both delete procedures.
- Create stored procedures that update records in the `publishers` and `titles` tables and an upper-level stored procedure that controls the execution of both update procedures.
- Grant appropriate permissions on all upper-level stored procedures.

*At the Primary Site:*

- Mark each upper-level stored procedure for replication, using **sp_setrepproc**.
- Create a function replication definition for each upper-level stored procedure.

*At the Replicate Site:*
Create subscriptions to the function replication definitions.

### Stored Procedures with Insert Clauses

Create the **ins_publishers** and **ins_titles** insert stored procedures and the upper-level stored procedure **ins_pub_title** using the sample scripts.

The insert procedures are identical at the primary and replicate sites. The upper-level stored procedure that controls the insert procedures and the insert procedures observe the following logic:

- A publisher record is inserted only when there is no title ID.
- A title record is inserted only when the publisher exists.

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure
 create procedure ins_publishers
 (@pub_id char(4), @pub_name varchar(40)=null,
 city varchar(20)=null, @state char(2)=null)
```

```
as
    insert publishers values (@pub_id,
    @pub_name, @city, @state)
/* end of script */
```

```
-- Execute this script at NY and SF data servers
 -- Creates stored procedure
 create procedure ins_titles
 (@title_id tid, @title varchar(80), @type char(12),
 @pub_id char(4)=null, @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime, @contract bit)
 as
 if not exists (select pub_id from publishers
         where pub_id=@pub_id)
     raiserror 20001 "** FATAL ERROR: Invalid publishers id **"
 else
     insert titles values (@title_id, @title, @type, @pub_id,
     @price,    @advance, @total_sales, @notes, @pubdate, @contract)
 /* end of script */
```

```
-- Execute this script at NY and SF data servers
 -- Creates stored procedure
 create procedure ins_pub_title
 (@pub_id char(4), @pub_name varchar(40)=null,
 @city varchar(20)=null, @state char(2),
 @title_id tid=null, @title varchar(80)=null, @type char(12)=null,
 @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime=null, @contract bit)
 as
 begin
     if @pub_name != null
     exec ins_publishers @pub_id, @pub_name, @city, @state
     if @title_id != null
     exec ins_titles @title_id, @title, @type, @pub_id, @price,
     @advance, @total_sales, @notes, @pubdate, @contract
 end/*
 end of script */
```

### Stored Procedures with Delete Clauses

Create the **del_publishers** and **del_titles** stored procedures and the upper-level stored procedure **del_pub_title** using the sample scripts.

The delete procedures are identical at the primary and replicate sites. The upper-level stored procedure that controls the delete procedures and the delete procedures observe the following logic:

- When a record is deleted, all dependent child records are also deleted.
- A publisher record is not deleted when a title record exists.

```
-- Execute this script at NY and SF data servers
 -- Creates stored procedure
 create procedure del_publishers
 (@pub_id char(4))
```

```
as
begin
if exists (select * from titles where pub_id=@pub_id)
    raiserror 20005 "**FATAL ERROR: Existing titles**"
else
    delete from publishers where pub_id=@pub_id
end
/* end of script */
```

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure /
create procedure del_titles
(@title_id tid, @pub_id char(4)=null)
as
if @pub_id=null
    delete from titles where title_id=@title_id
else
    delete from titles where pub_id=@pub_id
end
/* end of script */
```

```
-- Execute this script at NY and SF data servers
-- Creates stored procedure
create procedure del_pub_title
(@pub_id char(4), @state char(2), @title_id tid=null)
as
begin
    if @title_id != null
        begin
            exec del_titles @title_id
            return
        end
    if @pub_id != null
        begin
            exec del_titles @title_id, @pub_id
            exec del_publishers @pub_id
        end
end
/* end of script */
```

### Stored Procedures with Update Clauses

Create the **upd_publishers** and **upd_titles** stored procedures and the upper-level stored procedure **upd_pub_title** that controls the execution of **upd_publishers** and **upd_titles** using the sample scripts. The update procedures differ at the primary and replicate sites.

*At the Primary Site:*

Update procedures observe the following logic:

- Raise an error on an unknown `pub_id`.
- If a title does not exist, insert one.

The `upd_pub_title` stored procedure has an additional column, `old_state`, that enables replicates to subscribe to rows that migrate.

```
-- Execute this script at NY data servers
 -- Creates stored procedure
 create procedure upd_publishers
 (@pub_id char(4), @pub_name varchar(40),
 @city varchar(20), @state char(2))
 as
 if not exists
     (select * from publishers where pub_id=@pub_id)
     raiserror 20005 "**FATAL ERROR: Unknown publishers id**"
 else
     update publishers set
     pub_name=@pub_name,
     city=@city,
     state=@state
     where pub_id = @pub_id
 end
 /* end of script */
```

```
-- Execute this script at NY data servers
 -- Creates stored procedure
 create procedure upd_titles
 (@title_id tid, @title varchar(80), @type char(12),
 @pub_id char(4)=null, @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime, @contract bit)
 as
 if not exists
     (select * from titles where title_id=@title_id)
     raiserror 20005 "**FATAL ERROR: Unknown title id**"
 else
     update titles set
     title=@title,
     type=@type,
     pub_id=@pub_id,
     price=@price,
     advance=@advance,
     total_sales=@total_sales,
     notes=@notes,
     pubdate=@pubdate,
     contract=@contract
     where title_id = @title_id
 end
 /* end of script */
```

```
-- Execute this script at NY data server
 -- Creates stored procedure
 create procedure upd_pub_title
 (@pub_id char(4), @pub_name varchar(40)=null,
 @city varchar(20)=null, @state char(2)=null,
 @title_id tid=null, @title varchar(80)=null, @type char(12)=null,
 @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime=null, @contract bit, @old_state char(2))
 as
 begin
 if not exists (select * from publishers where pub_id=@pub_id)
```

```
    raiserror 20005 "**FATAL ERROR: Unknown publishers id**"
else
    exec upd_publishers @pub_id, @pub_name, @city, @state
if @title_id != null
begin
    if not exists
        (select * from titles where title_id=@title_id)
        exec ins_titles @title_id, @title, @type, @pub_id,
            @price, @advance, @total_sales, @notes, @pubdate,
            @contract
    else
    exec upd_titles @title_id, @title, @type, @pub_id, @price,
        @advance, @total_sales, @notes, @pubdate, @contract
end
end
/* end of script */
```

*At the Replicate Site:*
Update procedures observe the following logic:

- Raise an error on an unknown `pub_id`.
- If title does not exist, insert one.
- Implement correct update migration as shown in this table.

**Table 2. Migration Strategy for Replicate Site (CA)**

| Old State | New State | Update Procedure Must |
|---|---|---|
| CA | CA | Update `publishers` and `titles` tables normally. |
| CA | NY | Delete publisher and cascade delete of all titles associated with publisher. |
| NY | CA | Insert new publisher and title (if any). |

These scripts create the **upd_publishers** and **upd_titles** stored procedures and the managing stored procedure **upd_pub_title** that controls the execution of **upd_publishers** and **upd_titles**.

```
-- Execute this script at SF data servers
 -- Creates stored procedure
 create procedure upd_publishers
 (@pub_id char(4), @pub_name varchar(40),
 @city varchar(20), @state char(2))
 as
 if not exists
     (select * from publishers where pub_id=@pub_id)
     raiserror 20005 "**FATAL ERROR: Unknown publishers id**"
 else
     update publishers set
     pub_name=@pub_name,
     city=@city,
```

```
    state=@state
    where pub_id = @pub_id
end
/* end of script */
```

```
-- Execute this script at SF data servers
 -- Creates stored procedure
 create procedure upd_titles
 (@title_id tid, @title varchar(80), @type char(12),
 @pub_id char(4)=null, @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime, @contract bit)
 as
 if not exists
     (select * from titles where title_id=@title_id)
     exec ins_titles @title_id, @title, @type, @pub_id,
         @price, @advance, @total_sales, @notes, @pubdate,
         @contract
 else
     update titles set
     title=@title,
     type=@type,
     pub_id=@pub_id,
     price=@price,
     advance=@advance,
     total_sales=@total_sales,
     notes=@notes,
     pubdate=@pubdate,
     contract=@contract
     where title_id = @title_id
 end
 /* end of script */
```

```
-- Execute this script at SF data servers
 -- Creates stored procedure
 create procedure upd_pub_title
 (@pub_id char(4), @pub_name varchar(40)=null,
 @city varchar(20)=null, @state char(2),
 @title_id tid=null, @title varchar(80)=null, @type char(12)=null,
 @price money=null, @advance money=null,
 @total_sales int=null, @notes varchar(200)=null,
 @pubdate datetime=null,@contract bit, @old_state char(2))
 as
     declare @rep_state char (2)
 begin
     select @rep_state=state from publishers
     where pub_id=@pub_id

     if @old_state = @state
     begin
         exec upd_publishers @pub_id, @pub_name,@city, @state
         if @title_id != null
             exec upd_titles @title_id, @title, @type,
                 @pub_id, @price,@advance, @total_sales,
                 @notes, @pubdate, @contract
     end
```

```
    else if @rep_state = @old_state
        begin
            exec del_titles @title_id, @pub_id
            exec del_publishers @pub_id
        end
    else if @rep_state = null
        begin
            exec ins_publishers @pub_id, @pub_name, @city,
                @state
            if @title_id != null
                exec ins_titles @title_id, @title, @type,
                    @pub_id,@price, @advance, @total_sales,
                    @notes, @pubdate,@contract
        end
end
/* end of script */
```

### Function Replication Definitions

Create applied function replication definitions on the primary Replication Server for `ins_pub_title`, `del_pub_title`, and `upd_pub_title` using the sample scripts.

For inserts and deletes, only `state` is a searchable column; for updates, `old_state` is also searchable.

```
-- Execute this script at NY data servers
-- Creates replication definition ins_pub_title
create applied function replication definition ins_pub_title
with primary at MIAMI_DS.pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2),
 @title_id varchar(6),
 @title varchar(80),
 @type char(12),
 @price money,
 @advance money,
 @total_sales int,
 @notes varchar(200),
 @pubdate datetime,@contract bit)
searchable parameters (@state)
go
/* end of script */
```

```
-- Execute this script at NY data servers
-- Creates replication definition upd_pub_title
create applied function replication definition upd_pub_title
with primary at MIAMI_DS.pubs2
(@pub_id char(4),
 @pub_name varchar(40),
 @city varchar(20),
 @state char(2),
 @title_id varchar(6),
 @title varchar(80),
 @type char(12),
```

```
 @price money,
 @advance money,
 @total_sales int,
 @notes varchar(200),
 @pubdate datetime,
 @contract bit,
 @old_state char(2))
searchable parameters (@state, @old_state)
go
/* end of script */
```

### Subscriptions

Creates the subscriptions at the replicate Replication Server using the no-materialization method. Use this method when you do not need to load data at the replicate site.

To ensure proper subscription migration, you must create two subscriptions for `upd_pub_title`.

```
-- Execute this script at SF data servers
 -- Creates subscription for del_pub_title, ins_pub_title,
        and upd_pub_title
 create subscription del_pub_title_sub
 for del_pub_title
 with replicate at SF_DS.pubs2
 where @state='CA'
 without materialization
 go
```

```
create subscription ins_pub_title_sub
 for ins_pub_title
 with replicate at SF_DS.pubs2
 where @state='CA'
 without materialization
 go
```

```
create subscription upd_pub_title_sub1
 for upd_pub_title
 with replicate at SF_DS.pubs2
 where @state='CA'
 without materialization
 go
```

```
create subscription upd_pub_title_sub2
 for upd_pub_title
 with replicate at SF_DS.pubs2
 where @old_state ='CA'
 without materialization
 go
 /* end of script */
```

# Backup and Recovery Planning

Replication Server includes a variety of tools and methods you can use to return primary and replicate sites to a consistent state after a system component failure.

## Protection Against Data Loss

Replication Server runs in distributed database systems with many other hardware and software components, including Adaptive Servers and other data servers, Replication Agents, LANS, WANs, and client application programs.

Any of these components, including Replication Server, may occasionally fail. Replication Server is a fault-tolerant system, designed with this possibility in mind. During most failures, it waits for the failure to be corrected and then continues its work. When failures require restarting Replication Server or a Replication Agent, the start-up process guarantees that replication is resumed without loss or duplication of data.

Protecting against data loss is the same with a replication system as with a centralized database system. In both cases, there is one definitive version of the data, and you should invest considerable planning and resources to protect it.

In a replication system, if the primary data is protected, all replicate data can ultimately be recovered. A site can replace lost replicate data simply by re-creating its subscriptions.

If replicated transactions are lost at the primary site (as happens, for example, when the primary database is rolled back to a previous dump), consistency may be lost between primary and replicated data.

The backup and recovery methods available in a replication system include preventive and recovery measures.

Preventive measures include:

- Warm standby
- Hardware data mirroring (hot standby)
- Longer save intervals
- Coordinated dumps

Recovery measures include:

- Subscription initialization
- Subscription reconciliation utility (**rs_subcmp**)
- Database recovery
- Restoring coordinated dumps

- Database resynchronization

# Preventive Measures

Replication Server provides you with various preventive measures, which you can take to protect data in your replication system.

## Standby Applications

Protect the data in your replication system by maintaining separate (standby) copies of primary data.

Two possible standby methods are:

- Warm standby application – a pair of databases from the same vendor, one of which is a backup copy of the other. Client applications update the active database; Replication Server maintains the standby database by copying supported operations to the active database.
- Hardware data mirroring – a form of hot standby application. With the use of additional hardware, data mirroring maintains an exact copy of the primary data by reproducing *all* operations on the primary data.

### See also

- *Save Interval* on page 82
- *Coordinated Dumps* on page 82

### Compare Methods

Weigh the differences between a hot standby application and a warm standby application.

In a hot standby application, a standby database can be placed into service without interrupting client applications and without losing any transactions. A hot standby database guarantees that transactions committed on the active database are also committed on the standby. When both databases are running, the active database and the standby database are in sync, and the hot standby database is ready for immediate use.

Alternately, a warm standby application maintained by Replication Server:

- Can be used in environments where data mirroring applications cannot, especially when necessary hardware is not available.
- Tolerates temporary network failures better than some hot standby applications because committed transactions can be stored on the active database, even when the standby database is not running.
- Minimizes overhead on the active database because the active database does not need to verify that the databases are in sync.

However, a warm standby application maintained by Replication Server also:

- Requires some interruption of client applications when switching to the standby database.
- May not have executed in the standby database the most recent transactions committed in the active database.

### Warm Standby

A warm standby application is a pair of databases, one of which is a backup copy of the other. Client applications update the active database; Replication Server maintains the standby database as a copy of the active database.

The Replication Server warm standby application for Adaptive Server databases is described in the *Replication Server Administration Guide Volume 2 > Managing Warm Standby Applications*.

---

**Note:** Replication Server version 12.0 and later supports Sybase Failover available in Adaptive Server Enterprise version 12.0. Failover support is not a substitute for warm standby. While warm standby applications keep a copy of a database, Failover support accesses the same database from a different machine. Connections from Replication Server to warm standby databases work the same way.

For detailed information about how Failover support works in Replication Server, see *Replication Server Administration Guide Volume 2 > Configuring the Replication System to Support Sybase Failover* and *Replication Server Administration Guide Volume 2 > High Availability on Sun Cluster 2.2*.

---

### Hardware Data Mirroring

To ensure the highest data availability, you can mirror the most critical data in the replication system. Mirroring duplicates I/O operations, maintaning two identical copies of the data.

If the active media fails, the standby is brought online instantly. Mirroring all but eliminates the possibility of transaction loss.

The most beneficial places to use mirroring in a replication system are listed here in priority order:

1. Primary database transaction logs
   Transaction logs store transactions that have not been dumped to tape. If the primary transaction log is lost, transactions must be resubmitted.
2. Primary database
   A database can be recovered by reloading a previous database dump and subsequent transaction dumps. However, recovering a database that stores primary data also requires recovering or reinitializing the data that has been replicated throughout the enterprise. Extended downtime is often catastrophic for OLTP systems. Mirroring the primary data can prevent this type of catastrophe.
3. Replication Server stable queues
   Replication Server stores transactions in store-and-forward disk queues called stable queues. It allocates the queues from disk partitions assigned to the Replication Server using the **create partition** command.

**Note: create partition** makes a partition available to Replication Server, replacing the existing **add partition** command. **add partition** continues to be supported for backward compatibility. The syntax and usage of the two commands are identical. See *Replication Server Reference Manual > Replication Server Commands >* ***create partition***.

The data stored in stable queues is redundant; it originates in the primary database transaction log. However, if a stable queue is lost, Replication Server cannot deliver transactions to replicate sites. As a result, subscriptions at replicate sites must be reinitialized. Mirroring disk partitions protects stable queues and minimizes potential downtime for replicate databases.

4. Replication Server System Database (RSSD)

   Recovering from a failure of the RSSD can be a complex process if data such as replication definitions, subscriptions, routes, or function or error classes have been modified since the last backup. See *Replication Server Administration Guide Volume 2 > Replication System Recovery*.

   Mirroring the RSSD can prevent system data loss and the necessity of a complex recovery process. If you choose not to mirror the RSSD, be sure to back up the RSSD after any RCL operation that changes system data.

## Save Interval

You can configure a route from one Replication Server to another, or a connection from a Replication Server to a database, so that the Replication Server stores stable queue messages for a period of time after delivery to the destination. This period of time is called the save interval.

The save interval creates a backlog of messages at the source, which allows the replication system to tolerate a partition failure if it is corrected within the save interval. If the stable queues at the destination fail, you can rebuild them and have the source Replication Server resend the backlogged messages.

Refer to the *Replication Server Administration Guide Volume 2* for details.

## Coordinated Dumps

When you recoover a database by restoring a backup, you must also make replicated data in the affected databases at other sites consistent with the primary data. Replication Server let you coordinate database dumps and transaction dumps at all sites in a distributed system.

A database dump or transaction dump is initiated from the primary database. The Replication Agent retrieves the dump record from the log and submits it to the Replication Server so that the dump request can be distributed to the replicate sites. This guarantees that the data can be restored to a known point of consistency.

A coordinated dump can be used only with databases that store primary data or replicated data, but not both. It is initiated from within a primary database.

Refer to the *Replication Server Administration Guide Volume 2* for instructions on creating coordinated dumps.

**See also**

- *Standby Applications* on page 80
- *Save Interval* on page 82

# Recovery Measures

Replication Server provides you various options for recovering data that is lost after a component failure in a replication system.

## Re-create Subscriptions

Resolve all inconsistencies in the primary version of the data. One way to recover from a failure at a remote site is to re-create the subscriptions.

Re-creating subscriptions is most expedient for small replicated tables. Large subscriptions and primary data failures require more robust recovery methods.

## Subscription Reconciliation Utility ( rs_subcmp )

The **rs_subcmp** utility tests for rows that are missing, orphaned, or inconsistent in a replicate table and corrects the discrepancies.

Using **rs_subcmp** may be more appropriate for recovering from minor inconsistencies than a more disruptive recovery procedure such as a coordinated load. See *Replication Server Reference Manual* > ***rs_subcmp***.

## Database Recovery

When a primary database fails, all committed transactions can be recovered if the database and the transaction log are undamaged. If the database or the transaction log is damaged, you must load a database dump and transaction dumps to bring the database to a known state, and then resubmit the transactions that were executed after the last transaction dump.

When you run Replication Server and Replication Agents in recovery mode, you can replay transactions from reloaded dumps and make sure that all transactions in the primary database are replicated and that no transactions are duplicated. For more information about recovering primary databases from dumps, see the *Replication Server Administration Guide Volume 2*.

## Restoring Coordinated Dumps

Restoring database or transaction dumps created by the coordinated dump process returns the primary and replicated data to a previous, consistent state.

See the *Replication Server Administration Guide Volume 2*.

## Database Resynchronization

Database resynchronization allows you to rematerialize your replicate database and resume further replication without data loss or inconsistency, and without forcing a quiesce of your primary database.

Database resynchronization is based on obtaining a data dump from a trusted source and applying the dump to the target database you want to resynchronize.

See *Replication Server Administration Guide Volume 2 > Resynchronizing Replicate Databases* for general information and for the commands, parameters, procedures, and scenarios for database resynchronization for Adaptive Server. To resynchronize Oracle databases, see *Replication Server Heterogeneous Replication Guide > Resynchronizing Oracle Replicate Databases* and the Replication Agent documentation.

# Introduction to Replication Agents

A Replication Agent is a Replication Server client that retrieves information from the transaction log for a primary database and formats it for the primary Replication Server. RepAgent is the Replication Agent component for Adaptive Server.

Replication Agent detects changes to primary data and ignores changes to nonprimary data. Using Log Transfer Language (LTL), a subset of Replication Control Language (RCL), Replication Agent sends changes in primary data to the primary Replication Server, which distributes the information to replicate databases.

The Replication Agent connections status is derived from the status of the Replication Agent thread in the Replication Server, and the status of the Replication Agent process that is extracting data from the primary database. If either the Replication Agent thread or the Replication Agent process has stopped running, the RMS returns a suspended state. The RMS also returns a description if either of the components is not running.

Sybase provides Replication Agent components for other non-ASE data servers, including IBM DB2 Universal Database, Microsoft SQL Server, and Oracle. See the *Replication Server Heterogeneous Replication Guide* and the Replication Server Options documentation for the databases actively supported by Replication Server.

Replication Agent for DB2 is the Replication Agent component for the z/OS-based DB2 Universal Database. Sybase Replication Agent is the Replication Agent component for DB2 Universal Database (on UNIX and Windows platforms), Microsoft SQL Server, and Oracle.

RepAgent is an Adaptive Server thread. Replication Agent for DB2 is a separate process that resides on the z/OS host. Sybase Replication Agent is a separate application that resides on a UNIX or Windows host.

A Replication Agent performs:

1. Logs in to the Replication Server.
2. Sends a **connect source** command to identify the session as a log transfer source and to specify the database for which transaction information will be transferred.
3. Gets the name of the maintenance user for the database from the Replication Server. RepAgent filters out operations executed by the maintenance user, unless the **send_maint_xacts_to_replicate** or **send_warm_standby_xacts** configuration parameter is set to **true**.
4. Requests the secondary truncation point for the database from the Replication Server. This returns a value, called the *origin queue ID*, that RepAgent uses to find the location in the transaction log where it is to begin transferring transaction operations. The Replication Server has already received operations up to this location.
5. Retrieves records from the transaction log, beginning at the record following the secondary truncation point, and formats the information into LTL commands.

## Format of the Origin Queue ID

The origin queue ID is a unique 32-byte binary string that increases in value as each new log record is transferred. The value must increase, because Replication Server ignores records with an ID lower than the highest ID stored in the inbound queue.

When the Replication Server is restarted, it must be able to map the origin queue ID to the original log record so that it can send the next log record with an increased ID value.

A Replication Agent for data server other than Adaptive Server can generate origin queue IDs in any format, as long as the value increases and can be used to find the original log record. See the *Replication Server Heterogeneous Replication Guide* and Replication Server Options documentation for replication agent information for non-ASE actively supported data servers.

**Table 3. Format of the Origin Queue ID for Adaptive Server RepAgent**

| Bytes | Contents |
|---|---|
| 1–2 | Database generation number used for recovering after reloading coordinated dumps |
| 3–8 | Log page timestamp for the current record |
| 9–14 | Row ID of the current row. Row ID = page number (4 bytes) + row number (2 bytes) |
| 15–20 | Row ID of the begin record for the oldest open transaction |
| 21–28 | Date and time of the begin record for the oldest open transaction |
| 29–30 | An extension used by the RepAgent to roll back orphan transactions |
| 31–32 | Unused |

Bytes 21–28 contain an 8-byte `datetime` datatype value that is the time of the oldest partially transferred transaction in the database log. Replication Server prints the date and time in this field in a message that helps the database administrator locate the offline dumps needed for recovery. If you do not store a valid date in this field, the date and time printed in the message are meaningless. However, the message also contains the entire origin queue ID printed in hexadecimal format, so if you put the date and time in a location other than bytes 21–28, the replication system administrator can extract it from the origin queue ID.

## Replication Agent Products

Replication Agent products extend the capabilities of Replication Server by allowing non-Sybase database servers to serve as primary database servers in a Sybase replication system.

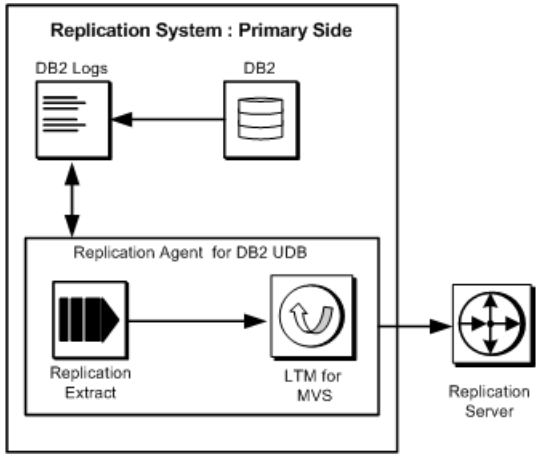
Sybase offers the following Replication Agent products for non-Sybase databases:

- Replication Agent for DB2
- Sybase Replication Agent

## Replication Agent for DB2

Replication Agent for DB2 captures database transactions in a DB2 primary database on an z/OS mainframe platform and sends them to Replication Server.

**Figure 23: Replication Agent Data Flow for DB2**




Replication Agent for DB2 fits into a replication system as follows:

- The primary database is DB2, which runs as a subsystem in z/OS. The transaction logs are DB2 logs.
- Replication Agent for DB2 provides a log extract, called Replication Extract, that reads the DB2 logs and retrieves the relevant DB2 active and archive log entries for tables marked for replication.
- LTM for MVS receives the data marked for replication from Replication Extract and transfers this data to Replication Server using the TCP/IP communications protocol.
- Replication Server then applies the changes to the replicate databases.

### DB2 Transaction Log

The DB2 database server logs changes to rows in DB2 tables as they occur. The information written to the transaction log includes copies of the data before and after the changes.

In DB2, these records are known as undo and redo records. Control records are written for commits and aborts. These records are translated to commits and rollbacks

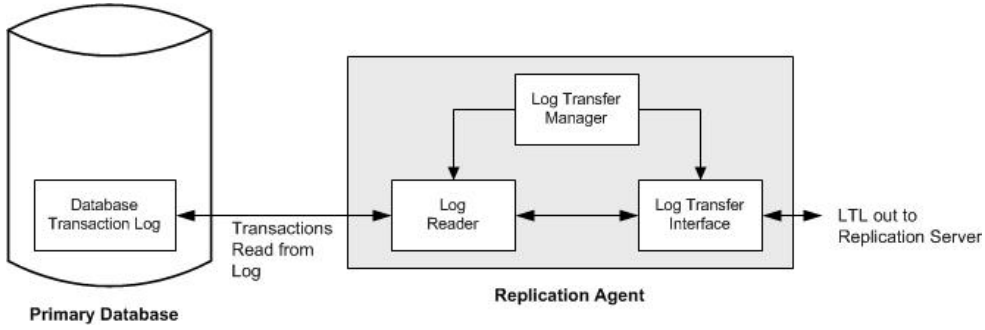
The DB2 log consists of a series of data sets. Replication Extract uses these log data sets to identify DB2 data changes. Since DB2 writes change records to the active log as they occur, Replication Extract can process the log records immediately after they are entered.

## Sybase Replication Agent

Sybase Replication Agent is a replication system component that captures transactions from a DB2 Universal Database (on UNIX and Windows platforms), Microsoft SQL Server, or Oracle primary database, and then transfers those transactions to Replication Server.

After transaction data is retrieved from the primary database, the Log Transfer Interface (LTI) component of Replication Agent processes the transaction and the resulting "change set" data and generates LTL output, which Replication Server uses to distribute the transaction to subscribing replicate databases.

**Figure 24: Sybase Replication Agent Data Flow**




Sybase Replication Agent uses information stored in the Replication Server System Database (RSSD) of the primary Replication Server to determine how to process the replicated transactions to generate the most efficient LTL.

After it receives LTL from Replication Agent, the primary Replication Server sends the replicated transaction to a replicate database, either directly or by way of a replicate Replication Server. The replicate Replication Server converts the replicated data to the native language of the replicate database, and then sends it to the replicate database server for processing. When the replicated transaction is processed successfully by the replicate database, the replicate database is synchronized with the primary database.

Sybase Replication Agent runs as a standalone application, independent of the primary database server, the primary Replication Server, and any other components of a replication system.

Sybase Replication Agent can reside on the same host machine as the primary database or any other component of the replication system, or it can reside on a separate machine from any other replication system components.

### Replication Agent Communications

Replication Agent uses a single instance of the Sybase JDBC driver (jConnect™ for JDBC™) to manage all of its connections to Open Client/Server applications, including the primary Replication Server and its RSSD.

In the case of the primary database server, Sybase Replication Agent connects to the JDBC driver for the primary database.

While replicating transactions, Replication Agent maintains connections with both the primary database and the primary Replication Server. In addition, Replication Agent occasionally connects to the RSSD of the primary Replication Server to retrieve replication definition data.

### Java Implementation

Sybase Replication Agent components are implemented in the Java programming language. Therefore, to run Sybase Replication Agent, you must have a Java Runtime Environment (JRE) installed on any computer that acts as the Replication Agent host machine.

## Replication Server and LTL Compatibility

Replication Agents and Replication Server negotiate the version of LTL to use for the session. This ensures that Replication Agents written for earlier versions of Replication Server remain compatible with more recent and current versions.

| Replication Server Version | LTL Version |
|---|---|
| 15.7.1 SP100 | 760 |
| 15.7.1 | 750 |
| 15.7 | 740 |
| 15.5 | 730 |
| 15.2 | 720 |
| 15.1 | 710 |
| 15.0 | 700 |
| 12.6 | 500 |
| 12.5 | 400 |
| 12.1 | 300 |
| 12.0 | 300 |
| 11.5 | 200 |

| Replication Server Version | LTL Version |
|---|---|
| 11.0 | 103 |
| 10.1.1 | 102 |
| 10.1 | 101 |
| 10.0 | 100 |

- If the Replication Agent uses Replication Server version 15.7.1 SP100 features, it must use LTL version 760 and connect to Replication Server version 15.7.1 SP100 or later.
- If the Replication Agent uses Replication Server version 15.7.1 features, it must use LTL version 750 and connect to Replication Server version 15.7.1 or later.
- If the Replication Agent uses Replication Server version 15.7 features, it must use LTL version 740 and connect to Replication Server version 15.7 or later.
- If the Replication Agent uses Replication Server version 15.5 features, it must use LTL version 730 and connect to Replication Server version 15.5 or later.
- If the Replication Agent uses Replication Server version 15.2 features, it must use LTL version 720 and connect to Replication Server version 15.2 or later.
- If the Replication Agent uses Replication Server version 15.1 features, it must use LTL version 710 and connect to Replication Server version 15.1 or later.
- If the Replication Agent uses Replication Server version 15.0 features, it must use LTL version 700 and connect to Replication Server version 15.0 or later.
- If the Replication Agent uses Replication Server version 12.6 features, it must use LTL version 500 and connect to Replication Server version 12.6 or later.
- If the Replication Agent uses Replication Server version 12.5 features, it must use LTL version 400 and connect to Replication Server version 12.5 or later.
- If the Replication Agent uses Replication Server version 12.x features, it must use LTL version 300 and connect to Replication Server version 12.x or later.
- If the Replication Agent uses Replication Server version 11.5 features, it must use LTL version 200 and connect to Replication Server version 11.5 or later.
- If the Replication Agent uses Replication Server version 11.0 features, it must use LTL version 103 and connect to Replication Server version 11.0 or later.
- If the Replication Agent uses Replication Server 10.1.1 features, it must use LTL version 102 and connect to Replication Server version 10.1.1 or later.
- If the Replication Agent uses Replication Server 10.1 features, it must use LTL version 101 and connect to Replication Server version 10.1 or later.
- If the Replication Agent uses only Replication Server 10.0 features, it can use LTL 100 and can connect to any Replication Server.

# Data Replication into Non-Adaptive Server Data Servers

Sybase offers several Open Server gateway application products that you can use to access non-Sybase database servers for a replicate database.

## Interface for Replication Server with Non-ASE Data Servers

Replication Server updates the replicate data stored in databases by submitting requests to data servers. Support for Adaptive Server is provided with Replication Server. If your database is managed by a data server other than Adaptive Server, you must provide an interface for Replication Server to use.

This interface includes:

- Sybase Enterprise Connect™ Data Access (ECDA), ExpressConnect for Oracle (ECO), or ExpressConnect for HANA DB (ECH) to receive instructions from Replication Server and apply them to the data server.
- A maintenance account that Replication Server can use to log in to the gateway (for example ECDA).
- A function-string class to use with the database. The function strings in the class tell Replication Server how to format requests for the data server.
- An error class and error action assignments to handle errors the data server returns to Replication Server via the gateway.
- An `rs_lastcommit` table in each database that has replicated data. Replication Server uses this table to keep track of the transactions that have been successfully committed in the database.
- An **rs_get_lastcommit** function call to retrieve the last transaction from each source primary database.

The non-ASE data server support design of Replication Server provides several components of this interface for actively supported database servers. The design provides function string classes, error classes and actions, user-defined datatypes, and connection profiles to create the necessary tables and procedures in the replicate database.

See the *Replication Server Administration Guide Volume 1* and the *Replication Server Configuration Guide* for your platform for more information about the non-ASE support feature. See the *Replication Server Heterogeneous Replication Guide* and the Replication Server Options documentation for the databases actively supported by Replication Server.

## Sybase Database Gateway Products

Sybase Enterprise Connect Data AccessConnect provides the Sybase middleware building blocks for connectivity between clients and enterprise data sources. Using ECDA simplifies the integration of non-Sybase replicate databases into a Sybase replication system.

You can use ECDA to connect directly to a database server. The DirectConnect component in ECDA acts as an Open Server gateway by translating the Open Client/Server protocol used by Replication Server to the native communication protocol used by the non-Sybase replicate database.

You can use ECDA to connect to:

- Microsoft SQL Server
- DB2 for z/OS
- UDB for Linux, UNIX, and Windows
- ODBC accessible data sources
- Oracle

**Note:** While ECDA is capable of accessing Oracle databases, you must use ExpressConnect for Oracle to replicate to Oracle databases.

See the *Replication Server Options* documentation.

## ExpressConnect for Oracle

ExpressConnect for Oracle (ECO) is a library that is loaded by Replication Server 15.5 or later for Oracle replication.

The advantages of ECO include:

- It does not require a separate server process for starting, monitoring, or administering.
- Since Replication Server and ECO run within the same process, no SSL is needed between them, and also there is no requirement to configure settings previously covered in the ECDA for Oracle global configuration parameters.
- Server connectivity is configured via Replication Server using the **create connection** and **alter connection** commands, thus there is no need to separately configure the equivalent to the ECDA for Oracle **connect_string** setting. See *Replication Server Reference Manual*.
- Configuration of settings equivalent to the ECDA for Oracle service-specific settings such as **text_chunksize**, **autocommit**, **array_size** is also not required, as these settings are automatically determined by Replication Server (in some cases based on the Replication Agent input) and communicated to ECO.

ECO includes certain features similar to ECDA for Oracle:

- Same set of datatype transformations.
- Language and charset conversion between Sybase data and Oracle data. In ECO, this is configured using the `map.cfg` file.
- Replication of empty strings in an ASE primary database to an Oracle replicate database, results in a string value of 1 or more (depending on whether the column is `varchar` or `fixed char width` datatype) blank spaces in Oracle.

ExpressConnect for Oracle requires only the `tnsnames.ora` file in order to establish location transparency. It does not require an `interfaces` file like ECDA for Oracle. You must specify the service name defined in the `tnsnames.ora` file for connection configuration.

See *ExpressConnect for Oracle Installation and Configuration Guide*.

## ExpressConnect for HANA DB

ExpressConnect for HANA DB (ECH) is a library that is loaded by Replication Server for HANA DB replication.

The advantages of ECH include:

- It does not require a separate server process for starting, monitoring, or administering.
- Server connectivity is configured with the Replication Server **create connection** and **alter connection** commands, and there is therefore no need for a separate configuration. See *Replication Server Reference Manual*.

A Replication Server database connection to HANA DB can be:

- secure, in which the connection uses the **hdbuserstore** key specified in the database connection, or
- standard, in which the login requires a user ID, password, and host name. A standard connection requires an entry in the interfaces file.

See *Replication Server Heterogeneous Replication Guide*.

## Maintenance User

Replication Server logs in as the maintenance user specified in **create connection** for the database.

The gateway can log in to the data server with the same login name, or it can use another login name. Login name must have the permissions needed to modify the replicate data.

# Function-String Class

The Replication Server managing a database requires a function-string class. Replication Server provides function-string classes for Adaptive Server, and with its non-ASE data server support features, Replication Server provides function-string classes for all actively supported data servers.

If you are replicating to a non-ASE data server that is not actively supported by Replication Server, you must create a function-string class for that data server. You can either:

- Create a function-string class that inherits function strings from a system-provided class, or
- Create all the function strings yourself.

Replication Server sends the gateway a command that it constructs by mapping runtime values into the function string supplied for the function. Depending on how the function string is written and the requirements of the data server, your gateway can pass the command directly to the data server or process it in some way before it sends the request to the data server. See the Replication Server Options documentation for the actively supported data servers.

**Note:** Replication Server 15.2 and later includes connection profiles preloaded with function-string classes for actively supported databases. See *Replication Server Administration Guide Volume 1 > Connection Profiles*.

See the *Replication Server Administration Guide Volume 2 > Customizing Database Operations* for a list of Replication Server system functions that your database gateway may need to process.

## Creating Function-String Classes Using Inheritance

Replication Server lets you share function-string definitions between function-string classes by creating relationships between classes using a mechanism called function-string inheritance.

The system-provided classes `rs_default_function_class` and `rs_db2_function_class` can serve as parent classes for derived classes that inherit function strings from the parent class. You can create a derived class in order to customize certain function strings for your data server while retaining all other function strings from the parent class.

Use the **create function string class** command to create a derived class from the parent class `rs_default_class` or `rs_db2_function_class` that inherits from the parent class. Create customized function strings only as needed.

**Note:** `rs_db2_function_class` does not support replication of `text` or `image` data. To enable replication of `text` or `image` data for DB2 databases, you must customize the **rs_writetext** function string using the RPC method through a gateway.

See *Replication Server Administration Guide Volume 2 > Customizing Database Operations*for a detailed discussion of function-string inheritance.

### Creating Distinct Function-string Classes

If you use a class that does not inherit from a system-provided class, you must create all function strings yourself, and add new function strings whenever you create a new table or function replication definition.

Use **create function string class** to create a new function-string class, and then create function strings for all of the functions with function-string-class scope.

You must create **rs_insert**, **rs_update**, and **rs_delete** function strings for each table you replicate in the database.

If you are replicating columns with `text` or `image` datatypes, you must create **rs_datarow_for_writetext**, **rs_get_textptr**, **rs_textptr_init**, and **rs_writetext** function strings for each `text` or `image` column. The function-string name must be the `text` or `image` column name for the replication definition.

The **rs_select** and **rs_select_with_lock** function strings are needed only if the database has the primary data for a replication definition.

## Error Class

An error class determines how Replication Server handles the errors that are returned by your gateway. Use the **create error class** command to create an error class for your gateway.

You can define error processing for data server errors with the Replication Server API. You can create an error class for a database and specify responses for each error that the data server returns.

**Note:** Replication Server 15.2 and later includes connection profiles pre-loaded with error classes for actively supported databases. See *Replication Server Administration Guide Volume 1 > Connection Profiles* and *Replication Server Administration Guide Volume 2 > Handling Errors and Exceptions > Default non-ASE Error Classes*.

Use the **assign action** command to tell Replication Server how to respond to the errors returned by your gateway.

**Table 4. Replication Server Actions for Data Server Errors**

| Action | Description |
|---|---|
| **ignore** | Assume that the command succeeded and that there is no error or warning condition to process. This action can be used for a return status that indicates successful execution. |

| Action | Description |
|---|---|
| **warn** | Log a warning message, but do not roll back the transaction or interrupt execution. |
| **retry_log** | Roll back the transaction and retry it. Use **configure connection** to set the number of retry attempts. If the error recurs after exceeding the retry limit, write the transaction into the exceptions log and continue, executing the next transaction. |
| **log** | Roll back the current transaction and log it in the exceptions log. Then continue, executing the next transaction. |
| **retry_stop** | Roll back the transaction and retry it. Use **configure connection** to set the number of retry attempts. If the error recurs after retrying, suspend replication for the database. |
| **stop_replication** | Roll back the current transaction and suspend replication for the database. This is equivalent to using **suspend connection**, and this is the default action.<br>Since this action stops all replication activity for the database, it is important to identify the data server errors that can be handled without shutting down the database connection and assign them another action. |

The default error action is **stop_replication**. If you do not assign another action to an error, Replication Server shuts down the connection to your gateway.

See the *Replication Server Reference Manual* for more information about **create error class** and **assign action**.

## rs_lastcommit Table

Each row in the *rs_lastcommit* table identifies the most recent committed transaction that was distributed to the database from a primary database. Replication Server uses this information to ensure that all transactions are distributed.

The `rs_lastcommit` table should be updated by the **rs_commit** function string before the transaction is committed. This guarantees that the table is updated with every transaction Replication Server commits in the database.

Replication Server maintains the `rs_lastcommit` table as the maintenance user for the database. You must make sure that the maintenance user has all of the permissions needed for the table.

**Table 5. rs_lastcommit Table Structure**

| Column name | Datatype | Description |
|---|---|---|
| origin | int | An integer assigned by Replication Server that uniquely identifies the database where the transaction originated |
| origin_qid | binary(36) | The origin queue ID for the commit record in the transaction |
| secondary_qid | binary(36) | A queue ID for a stable queue used during subscription materialization |
| origin_time | datetime | (Optional column) Time at origin for the transaction |
| dest_commit_time | datetime | (Optional column) Time the transaction was committed at the destination |

The `origin` column is a unique key for the table. There is one row for each primary database whose data is replicated in this database.

If you use a coordinated dump with the database, you should update `rs_lastcommit` with the **rs_dumpdb** and **rs_dumptran** function strings.

For Adaptive Server databases, the **rs_commit**, **rs_dumpdb**, and **rs_dumptran** function strings execute a stored procedure named **rs_update_lastcommit** to update the `rs_lastcommit` table. This is the text of that stored procedure:

```
/* Create a procedure to update the
 ** rs_lastcommit table. */
 create procedure rs_update_lastcommit
     @origin int,
     @origin_qid binary(36),
     @secondary_qid binary(36),
     @origin_time datetime
 as
     update rs_lastcommit
         set origin_qid = @origin_qid,
         secondary_qid = @secondary_qid,
         origin_time = @origin_time,
         commit_time = getdate()
     where origin = @origin
     if (@@rowcount = 0)
     begin
         insert rs_lastcommit (origin,
             origin_qid, secondary_qid,
             origin_time, commit_time,
             pad1, pad2, pad3, pad4,
             pad5, pad6, pad7, pad8)
         values (@origin, @origin_qid,
             @secondary_qid,@origin_time,
             getdate(), 0x00, 0x00, 0x00,
             0x00, 0x00, 0x00, 0x00, 0x00)
```

```
    end
go
```

**Note:** Replication Server 15.2 and later creates the `rs_lastcommit` table for actively supported non-ASE data servers when you initially create a connection using a connection profile.

See the reference pages for the **rs_commit**, **rs_dumpdb**, and **rs_dumptran** functions in the *Replication Server Reference Manual* for more information.

## rs_get_lastcommit Function

Replication Server sends an **rs_get_lastcommit** function call to the gateway to retrieve the last transaction committed in the database from each source primary database.

The function string for **rs_get_lastcommit** can execute a simple **select**:

```
select origin, origin_qid, secondary_qid
    from rs_lastcommit
```

**Note:** The **rs_get_lastcommit** function is predefined in connection profiles for actively supported non-ASE data servers.

# International Replication Design Considerations

Replication Server and Replication Manager (RM), the Sybase Central plug-in for managing your replication system, both support international environments.

Replication Server and Replication Manager (RM):

- Localize messages into several languages
- Support for all Sybase-supported character sets, with character set conversion between Replication Server sites.
- Support for nonbinary sort orders

When you design a replication system for an international environment, understand the impact that language, character set, and sort order settings have on your system. Replication Server and the RM provide great flexibility in the configuration of these settings.

## Message Language

Replication Server allows you to print messages to the error log and to clients in several languages.

The language you choose must be compatible with the chosen character set. English, is compatible with all Sybase character sets, is the default language. See the *Replication Server Configuration Guide* for your platform for a list of supported languages.

Each server program in your replication system, including Replication Server, Adaptive Server, and other data servers, writes messages to its error log in the configured language. However, whether messages are sent to a client in the client's language depends on the server.

For example, Adaptive Server checks for the language setting of its client (Replication Server) and returns messages in that language. RepAgent, an Adaptive Server thread, also returns messages in the client language.

However, Replication Server does not check for a client's language; instead, it returns messages to clients in their own language. Thus, error logs can contain messages in different languages if the servers are configured in different languages.

**Note:** To avoid the confusion that can result from a mixed-language error log, configure the same language setting for all servers and clients at a given site.

## Changing the Replication Server Message Language

Configure the Replication Server message language.

**Note:** Because RepAgent automatically returns messages in the Replication Server language, you do not have to set a language parameter for RepAgent.

1. Shut down the Replication Server.
2. Using a text editor, change the value of **RS_language** in the Replication Server configuration file.
3. Restart Replication Server.

# Character Sets

Replication Server supports all Sybase-supported character sets and performs character set conversion of data and identifiers between primary and replicate sites.

Character sets must be compatible for character-set conversion to be successful. For example, single-byte character set data cannot be converted to a multibyte character set. For details about character-set compatibility, see the *Adaptive Server Enterprise System Administration Guide Volume 1*.

Your choice of a character set for a given server is influenced by the languages the server supports, the hardware and operating system it runs on, and the systems with which it interacts.

These things are true of Sybase-supported character sets:

- They are all supersets of 7-bit ASCII.
- Some are completely incompatible with each other, meaning that they have no characters in common beyond 7-bit ASCII.
- Among compatible character sets, some characters are not common to both sets—that is, no two character sets have all the same characters.

To change the default character set, see "Changing the Character Set and Sort Order." Although changing the default character set requires only changing the value of the **RS_charset** parameter in the Replication Server configuration file, follow the task steps exactly in the procedure to ensure that no replicate data is corrupted by the change.

## Character Set Conversion

Character-set conversion takes place at the destination Replication Server. Every message packed for the Replication Server Interface (RSI) includes the name of the source Replication Server character set. The destination Replication Server uses this information to convert the character data and identifiers to its own character set.

When it attempts character-set conversion, Replication Server checks whether the character sets are compatible. If they are incompatible, no conversion occurs. If they are compatible and

one or more characters that are not common to both sets are encountered, a ? (question mark) is substituted for each unrecognized characters.

In the Replication Server, context determines whether a ? is substituted or a character-set conversion exception is raised. For example, if Replication Server detects an incompatibility in a character being replicated, it substitutes a ? for the character; if it detects an incompatibility when converting the configuration file parameters, it prints an error message and shuts down.

You can use the **dsi_charset_convert** configuration parameter to specify whether or not Replication Server attempts character-set conversion. See *Replication Server Reference Manual* > ***configure connection***.

### See also

- *Subscriptions* on page 103
- *Changing the Character Set or Sort Order* on page 107

## Unicode UTF-8 and UTF-16 Support

Replication Server supports the default character set Unicode UTF-8 and three Unicode datatypes—`unichar`, `univarchar`, and `unitext`—which are UTF-16 encoded. Unicode allows you to mix different languages from different language groups in the same data server.

See the *Adaptive Server Enterprise System Administration Guide Volume 1*.

### UTF-8

UTF-8 (UCS Transformation Format, 8-bit form) is an international character set that supports more than 650 languages. UTF-8 is a variable-length encoding of the Unicode standard using 8-bit sequences.

UTF-8 supports all ASCII code values, from 0 to 127, as well as values from many other languages. Each nonsurrogate code value is represented in 1, 2, or 3 bytes. Code values beyond the basic multilingual plane (BMP) require surrogate pairs and 4 bytes.

Adaptive Server, Oracle, IBM UDB, and Microsoft SQL Server data servers all support UTF-8.

### UTF-16

UTF-16 (UCS Transformation Format, 16-bit form) is a fixed-length encoding of the Unicode standard using 16-bit sequences, where all characters are 2 bytes long. As with UTF-8, code values beyond the BMP are represented using surrogate pairs that require 4 bytes.

Both Replication Server and Adaptive Server encode three character datatype values in UTF-16:

- `unichar` – fixed-width Unicode.

- `univarchar` – variable-width Unicode.
- `unitext` – variable-width Unicode large object datatype introduced with ASE 15.0 and Replication Server 15.0. `unitext` can hold up to 1,073,741,823 Unicode characters or the equivalent of 2,147,483,647 bytes.

#### Requirements

To use the Unicode UTF-8 default character set or the `unichar` and `univarchar` datatypes, you must be running Replication Server version 12.5 or later and have set the site version to 12.5 or later.

The `unitext` datatype is fully supported if you have a site version and route version of 15.0 or later for the primary and replicate Replication Servers, and the LTL version is 700. If the LTL version is less than 700 at connect-source time, RepAgent and other Sybase Replication Agents convert `unitext` columns to `image`.

### Guidelines for Using Character Sets

Sybase strongly recommends that all servers at a given Replication Server site use the same character set, and that all of the Replication Servers in your replication system use compatible character sets.

To minimize character-set conversion problems:

- Use 7-bit ASCII (if possible) for all data servers, data, and object names.
  If your data and object names are all 7-bit ASCII or if all of your data servers and Replication Servers use the same character set, character set conversion will not present problems.
- If you need to replicate data between a single-byte and a multibyte server, restrict character data and object names to 7-bit ASCII to avoid corruption. Otherwise, you may experience problems. For example, Replication Server does not restrict server names to 7-bit ASCII, but Adaptive Server or the Connectivity Libraries may do so.
- When replicating between servers with different but compatible character sets (for example, ISO_1 and CP850), make sure that object names and character data do not include any 8-bit characters that are not common to both character sets.

## Sort Order

Replication Server uses sort orders, or collating sequences, to determine how character data and identifiers are compared and ordered. Replication Server supports all Sybase-supported sort orders, including non-binary sort orders. Non-binary sort orders are necessary for the correct ordering of character data and identifiers in European languages.

To change the default or Unicode sort order, see “Changing the Character Set and Sort Order.” Although changing the default sort order requires only changing the value of the **RS_sortorder** parameter in the Replication Server configuration file, follow the steps provided in the procedure to ensure that no replicate data is corrupted by the change.

**Note:** To make sure that data and identifiers are ordered consistently across your replication system, configure all Replication Server components with the same sort order .

### See also

- *Changing the Character Set or Sort Order* on page 107

## Subscriptions

Sort order and character sets must be consistent everywhere for subscriptions to be valid.

Subscriptions involve comparisons of data at the:

- Primary data server during materialization
- Primary Replication Server during resolution
- Replicate Replication Server during initialization and dropping
- Replicate data server during dropping

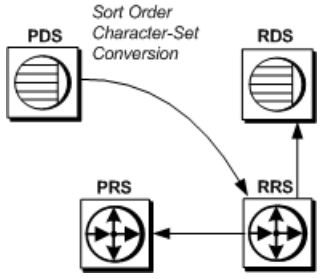
### See also

- *Changing the Character Set or Sort Order* on page 107

### Subscription Materialization

Materialization is creating and activating subscriptions, and copying data from the primary database to the replicate database, thereby initializing the replicate database.

**Figure 25: Subscription Materialization**




During subscription materialization:

- The replicate Replication Server logs in to the primary data server and issues a **select** statement to retrieve the primary data.
- The primary data server converts all character data to the replicate Replication Server character set. The replicate Replication Server character set, if it is different, must be installed at the primary data server.
- The replicate Replication Server inserts the data at the replicate data server.

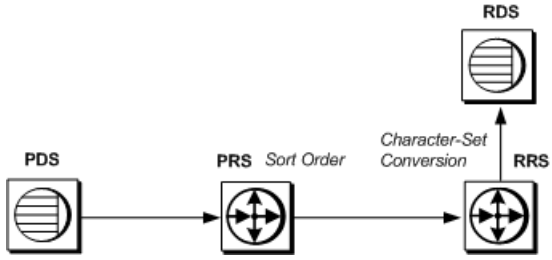
**Note:** In bulk materialization, a subscription is initialized by a user-chosen mechanism outside the replication system. Therefore, make sure that the initial data selection at the

primary data server uses the correct sort order and that the character data is converted to the replicate data server character set, if need be.

### Subscription Resolution

Subscriptions pass through phases before they are fully set up in your replication system.

**Figure 26: Subscription Resolution**



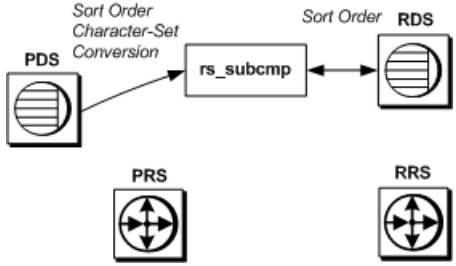

During subscription resolution:

- Adaptive Server RepAgent thread scans the log for updates.
- The primary Replication Server uses its sort order to determine what rows qualify for the subscription. The primary Replication Server also adds the name of the primary Replication Server character set to the RSI message.
- The replicate Replication Server converts the data to its character set, if necessary, and applies updates to the replicate data server.

### Subscription Reconciliation

Subcription reconciliation is for recovering inconsistency in a replicate table and correcting the discrepancies.

**Figure 27: Subscription Reconciliation**




During subscription reconciliation:

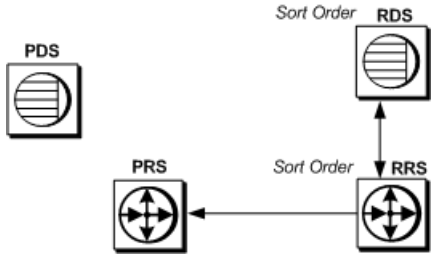
- **rs_subcmp** connects to the primary data server and to the replicate data server using the replicate data server's character set.

- The primary data server converts all character data to the replicate data server's character set (all **rs_subcmp** operations are performed in the replicate data server's character set). The replicate data server's character set, if it is different from the primary data server's character set, must be installed at the primary data server.
- **rs_subcmp** sends a **select** statement to both data servers. The sort order of each data server must be the same for this process to produce sensible results.

### Dematerialization

Dematerialization is when you execute a command to drop subscriptions at the replicate Replication Server.

**Figure 28: Subscription Dematerialization**




During subscription dematerialization:

- The replicate Replication Server selects data from the replicate data server to construct the dematerialization queue. The replicate data server uses its sort order to select the rows.
- The replicate Replication Server uses its sort order to throw out rows belonging to other subscriptions.
- The replicate Replication Server deletes the remaining rows from the replicate database.

## Unicode Sort Order

The Unicode sort order is different from the Replication Server sort order, and must be set independently.

To set the Unicode sort order, use a text editor to add the following line to the Replication Server configuration file:

```
RS_unicode_sort_order=unicode_sort_order
```

Make sure that you suspend the connection to the data server and shut down Replication Server before changing the Unicode sort order.

To change the current sort order, use the procedure described in "Changing the character set and sort order."

**Table 6. Supported Unicode Sort Orders**

| Name | Description |
|---|---|
| defaultml | UTF-16 default ML |
| altnoacc | CP850 alt: no accent |
| altdict | cp850 alt: lowercase first |
| altnocsp | CP850 alt: no case preference |
| scandict | CP850 Scandinavian dictionary |
| scannocp | CP850 Scandinavian, no case preference |
| binary | (Default) UTF-16 binary |
| dict | Latin-1 English dictionary |
| nocase | Latin-1 English, no case |
| nocasep | Latin-1 English, no case preference |
| noaccent | Latin-1 English, no accent |
| espdict | Latin-1 Spanish dictionary |
| espnocs | Latin-1 Spanish no case |
| espnoac | Latin-1 Spanish, no accent |
| rusnocs | 8859-5 Russian, no case |
| cyrnocs | 8859-5 Cyrillic, no case |
| elldict | 8859-7 Greek dictionary |
| hundict | 8859-2 Hungarian dictionary |
| hunnoac | 8859-2 Hungarian, no accents |
| hunnocs | 8859-2 Hungarian, no case |
| turknoac | 8859-9 Turkish, no accent |
| turknocs | 8859-9 Turkish, no case |
| thaidict | CP874 Thai dictionary |
| utf8bin | UTF-16 ordering matching UTF-8 |

You can also specify a Unicode sort order for **rs_subcmp**. See *Replication Server Reference Manual* > ***rs_subcmp***.

### See also

- *Changing the Character Set or Sort Order* on page 107

# Character Set and Sort Order

If you change the character set or sort order of Adaptive Server, you must also change the character set or sort order of each Replication Server that manages replication for the server and each associated RSSD that resides on a separate Adaptive Server.

If you change the sort order of Adaptive Server, you must also change the sort order of the replicate Replication Server and replicate data server to ensure that subscriptions are processed consistently.

After changing the character set or sort order, subscription semantics may change. Sort order changes can have obvious consequences. Suppose a subscription contains the clause "where last_name = MacGregor." If the sort order is changed from dict to binary, for example, "MacGregor" no longer qualifies for sorting.

*Synchronize the Primary and Replication Databases*

After you change the character set or sort order, make sure that the primary and replicate databases are resynchronized. Sybase recommends that you use one of these methods:

- Use **rs_subcmp** after changing the character set or sort order, or
- Purge all subscriptions before changing the character set or sort order, and then rematerialize all subscriptions after changing the character set or sort order.

**Note:** Use the purge and rematerialize method if any subscriptions contain character clauses. Only this method ensures that subscriptions with character clauses are resynchronized.

## Changing the Character Set or Sort Order

Change the character set or sort order in Replication Server.

All replicated transactions originating from Adaptive Server must arrive at the replicate data server before you can modify the character set or sort order. In addition to changing the sort order or character set, this procedure ensures that no data corruption occurs resulting from changing the character set or sort order.

1. Identify all Replication Servers and Adaptive Servers that are associated with the primary Adaptive Server—including all RSSDs for associated Replication Servers.

   If you are changing the character set: look up the character set of all Adaptive Servers in the Replication Server domain to ascertain if their character sets must also be changed. Sybase supports alternate character sets for servers in the same domain, but the implication for users is significant.

   If you are changing the sort order: Sybase recommends that all data servers in the Replication Server domain share the same sort order. This ensures that data and identifiers are ordered consistently throughout the replication system.

2. Quiesce all primary updates and make sure that they have been processed by Replication Server.

   If you are changing the character set, make sure that there are sufficient empty transactions to span a page in Adaptive Server. This ensures that after the Adaptive Server transaction log is emptied (see step 9), no data remains in the old character set.
3. Quiesce all associated Replication Servers.
4. Shut down all associated Replication Servers, RepAgents, and Replication Agents.
5. Change the character set and sort order in configuration files for the Replication Servers and, if applicable, for the Replication Agents.
6. Follow Adaptive Server procedures for changing the default character set and sort order of each associated Adaptive Server. See *Adaptive Server Enterprise System Administration Guide Volume 1 > Configuring Character Sets, Sort Orders, and Languages*.
7. Shut down all associated Adaptive Servers.
8. Start the associated Adaptive Servers in single-user mode—unless you can guarantee that there will be no activity at the primary and replicate databases.
9. Remove the secondary truncation point from each associated Adaptive Server. This step allows Adaptive Server to truncate log records that the RepAgent has not yet transferred to the Replication Server. From the Adaptive Server, enter:

   ```
   dbcc settrunc('ltm', 'ignore')
   ```
10. Truncate the transaction log. From the Adaptive Server, enter:

    ```
    dump transaction db_name with truncate_only
    ```
11. Reset the secondary truncation point. From the Adaptive Server, enter:

    ```
    dbcc settrunc('ltm', 'valid')
    ```
12. Reset the locator value for the primary database to zero. This step instructs Replication Server to get the new secondary truncation point from Adaptive Server and set the locator to that value. From the Adaptive Server, enter:

    ```
    rs_zeroltm data_server, db_name
    ```
13. Shut down and restart Adaptive Server in normal mode.
14. Restart the associated Replication Servers.
15. Allow RepAgents (or Replication Agents) to reconnect to Replication Servers by resuming log transfer. From the Replication Server, enter:

    ```
    resume log transfer from data_server.db_name
    ```
16. Start RepAgents.
17. Restart replication.

### When Changing the Character Set Changes the Character Width

If the character set change involves a change of character width, reload the stored procedure messages of all databases controlled by the Replication Server.

The stored procedure messages are in `$SYBASE/$SYBASE_REP/scripts/rsspmsg1.sql` and `rsspmsg2.sql` for UNIX, and `%SYBASE%\%SYBASE_REP%\scripts\rsspmsg1.sql` and `rsspmsg2.sql` for Windows.

For UNIX

When the change is from a single-byte to a multibyte character set:

```
isql -Uuser_name -Ppassword -Srssd_name -Jeucjis
< $SYBASE/$SYBASE_REP/scripts/rsspmsg2.sql
```

When the change is from a multibyte to a single-byte character set:

```
isql -Uuser_name -Ppassword -Srssd_name -Jiso_1
< $SYBASE/$SYBASE_REP/scripts/rsspmsg1.sql
```

For Windows

When the change is from a single-byte to a multibyte character set:

```
isql -Uuser_name -Ppassword -Srssd_name -Jeucjis
< %SYBASE%\%SYBASE_REP%\scripts\rsspmsg2.sql
```

When the change is from a multibyte to a single-byte character set:

```
isql -Uuser_name -Ppassword -Srssd_name -Jiso_1
< %SYBASE%\%SYBASE_REP%\scripts\rsspmsg1.sql
```

When the change is to the UTF-8 character set, install both `rsspmsg1.sql` and `rsspmsg2.sql`

## Summary

Provides a summary of topics discussed about replication designs.

- Replication Server can be configured to print messages to error logs and to clients in English, French, German, and Japanese. English is the default language.
- Sybase recommends that all servers at a replication site be configured with the same language.
- Replication Server supports all Sybase-supported character sets and sort orders, including non-binary sort orders and the Unicode UTF-8 character set.
- Replication Server performs character-set conversion of data and identifiers between primary and replicate Replication Servers and databases.
- Sybase recommends that all servers at a replication site use the same character set and that all Replication Servers in your system use compatible character sets.

- Sort order plays an important role in processing subscriptions, and it must be consistent everywhere for subscriptions to be valid.

# Capacity Planning

Plan the CPU, memory, disk, and network resources for your replication system.

A replication system consists of Replication Servers, Replication Agents (RepAgent or other Replication Agent), Replication Manager (RM), Replication Monitoring Services (RMS), and data servers.

**Warning!** As versions of Adaptive Server change, Replication Server capacity planning may change as a consequence of new Adaptive Server transaction log space requirements.

All capacity planning assumes that Adaptive Server is using a 2KB page size. If you are using larger page sizes, you must recalculate your space utilization needs to accommodate the Adaptive Server larger page size.

## Replication Server Requirements

Replication Sever requirements.

The minimum requirements for each Replication Server include:

- One Replication Agent for the Replication Server System Database (RSSD) if there will be a route from this Replication Server.
- At least one 20MB raw partition or operating system file for stable queues.
- An Adaptive Server for the RSSD or a SQL Anywhere data server (SA) for the ERSSD.
  An Adaptive Server for the RSSD must have:
  - At least 10MB of free device space for the RSSD database directory
  - Another 10MB of free device space for the RSSD transaction log directory

  A SQL Anywhere data server used as an ERSSD must have:
  - At least 5MB of free device space for the ERSSD database directory
  - At least 3MB of free device space for the ERSSD transaction log directory
  - Another 12MB of free device space for the ERSSD backup directory
- 20 user connections for the RSSD, in addition to the number of user connections needed by Adaptive Server users. When Replication Server starts, several threads simultaneoulsy attempt to read the RSSD. To accommodate this demand, increase the number of user connections by 20.
- One RSSD user connection for each RM and for each RMS in the replication system and one user connection per data server for each RM process and for each RMS process.
- Two user connections for each database containing primary data.
- One user connection for each replicate-only database.

- At least 512MB of RAM for the Replication Server executable program and all the Replication Agents, plus data and stack memory. (RepAgent is an Adaptive Server thread; it does not require any Replication Server memory.)

## Replication Server Requirements for Primary Databases

Specifies the Replication Server requirements for the primary database.

For each primary database it manages, a Replication Server needs:

- A RepAgent thread for Adaptive Server databases or other Replication Agent for non-Sybase databases
- One inbound stable queue
- One outbound stable queue
- One connection to the data server for the Data Server Interface (DSI)

See the release bulletin for more information about Adaptive Server compatibility requirements.

**Note:** If you are using a Replication Agent with a non-Sybase data source, see the appropriate Replication Agent documentation for information about compatibility requirements.

## Replication Server Requirements for Replicate Databases

Specifies the Replication Server requirements for the replicate database.

For each replicate database it manages, a Replication Server needs:

- One outbound stable queue
- One connection to the data server for the DSI.

## Replication Server Requirement for Routes

Specifies the Replication Server requirements for routes.

For each direct route to another Replication Server, a Replication Server needs:

- One outbound stable queue

## Data Volume (Queue Disk Space Requirements)

The most significant components in estimating the amount of resources required by the replication system are the volume of the data being replicated and the rate at which it is being updated.

To accurately calculate data volume, know these things about your replication system:

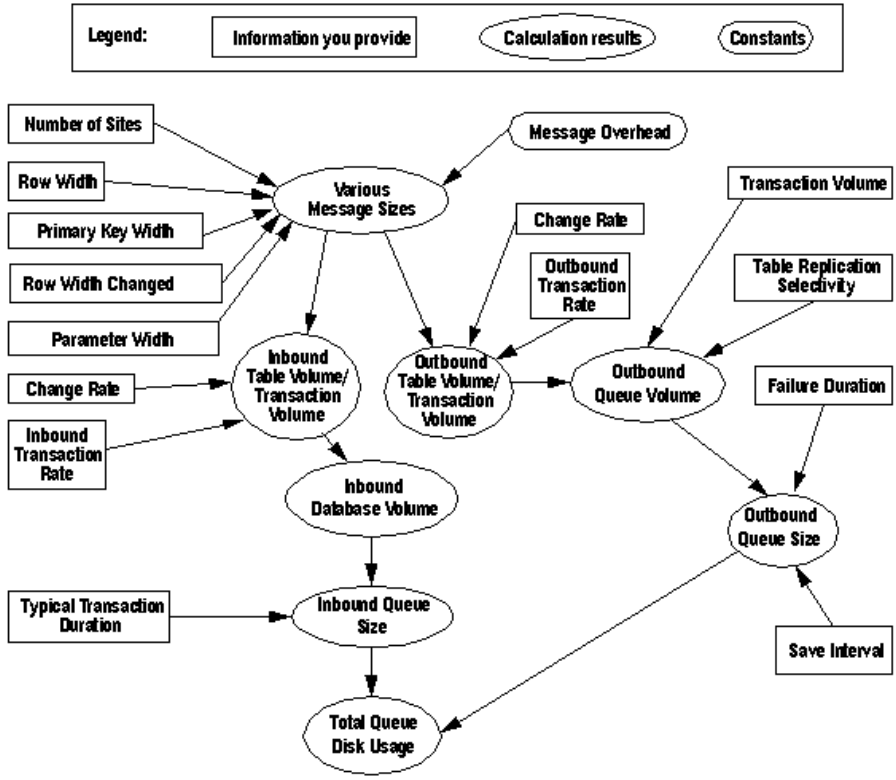
- The number of sites
- The widths of the rows in replicated tables
- The widths of parameters in replicated functions
- The number of modifications per second
- The duration of a typical transaction
- The selectivity of replication for replicated tables (the fraction of that table replicated through the queue)
- The length of time you want queues to hold transactions when a destination is unavailable

There are formulas for calculating queue size requirements. You can also use the **rs_fillcaptable** and **rs_capacity** procedures in an RSSD to get a queue size estimate. See the *Replication Server Reference Manual* for information about these stored procedures.

## Overview of Disk Queue Size Calculation

A graphical representation that illustrates the sequence and flow of calculating data volume and queue disk usage.

**Figure 29: Calculating Queue Disk Usage**




## Change Rate (Number of Messages)

The change rate is the maximum number of modifications (**inserts**, **deletes**, and **updates**) made to the table per time period. Change rate varies from table to table.

When planning capacity, always use a change rate greater than the maximum change rate recorded for that table.

**Note:** Figures for change rates are expressed as operations per unit of time (for example, 5 updates per second). Always use the same time period when calculating rates. If you measure updates per second, then you must use seconds as the basis for all other time period calculations.

## Change Volume (Number of Bytes)

The change volume is the amount of data changes in the table per time period. This amount varies; it is a function of the change rate and data size for each table.

## Calculating Table Volume

The volume of data being replicated is the size of each message times the number of messages replicated per second.

If you know the message size of each modification to a table and of each replicated function, calculate the total volume of messages generated in a database by adding the volumes of the individual tables and replicated functions.

### Table Volume Upper Bound Method

Calculate table volumes and database volumes for inbound and outbound queues separately. Or, calculate the volume only for outbound queues and use that figure as an approximation for both inbound and outbound volumes.

Generate the upper bound for each table by multiplying the update rate on the table by the size of the longest message. This gives you the upper bound of the table volume.

```
InboundTableVolumeupper = (Max[InboundMsgSize] *
    ChangeRate)
```

```
OutboundTableVolumeupper = (Max[OutboundMsgSize] *
    ChangeRate
```

where:

- *Max[InboundMsgSize]* and *Max[OutboundMsgSize]* are the sizes, in bytes, of the largest inbound and outbound message (typically, the message size with the largest parameters).
- *ChangeRate* is the maximum number of modifications to the table per time period.

### Table Volume Sum of Values Method

Obtain a precise calculation of data volume by adding together all of the volumes for each message type.

To calculate *InboundTableVolume*:

```
InboundTableVolume =
     (InboundMsgSizeupdate * ChangeRateupdate) +
     (InboundMsgSizeinsert * ChangeRateinsert) +
     (InboundMsgSizedelete * ChangeRatedelete) +
     (InboundMsgSizefunction1 * ChangeRatefunction1) +
     (InboundMsgSizefunction2 * ChangeRatefunction2) +
     ...
```

To calculate *OutboundTableVolume*:

```
OutboundTableVolume =
     (OutboundMsgSizeupdate * ChangeRateupdate) +
     (OutboundMsgSizeinsert * ChangeRateinsert) +
```

```
    (OutboundMsgSizedelete * ChangeRatedelete) +
    (OutboundMsgSizefunction1 * ChangeRatefunction1) +
    (OutboundMsgSizefunction2 * ChangeRatefunction2) +
    ...
```

where:

- *ChangeRate* is the maximum number of modifications to the table per time period. *ChangeRate* $_{update}$ refers to data updates, *ChangeRate* $_{insert}$ refers to data insertions, and so forth.
- *InboundMsgSize* and *OutboundMsgSize* are the different types of maximum inbound and outbound message sizes.

### See also

- *Message Sizes* on page 126

### Transaction Volume

*Transaction Volume* estimates data volume generated by the begin and commit records.

The formula for *InboundTransaction Volume* is:

```
InboundTransactionVolume = (MsgSizecommit + MsgSizebegin) *
InboundTransactionRate
```

where:

- *MsgSize* $_{begin}$ + *MsgSize* $_{commit}$ equals 450 bytes per transaction.
- *InboundTransactionRate* is the total number of transactions per time period at the primary database. This includes all transactions—those that update replicated tables and those that do not. One transaction may include one or more modifications to one or more tables.

The formula for *OutboundTransaction Volume* is:

```
OutboundTransactionVolume = (MsgSizecommit + MsgSizebegin) *
OutboundTransactionRate
```

where:

- *MsgSize* $_{begin}$ + *MsgSize* $_{commit}$ equals 450 bytes per transaction.
- *OutboundTransactionRate* is the total number of replicated transactions per time period replicated through a particular outbound queue. Each transaction may include one or more modifications to one or more tables.

  *OutboundTransactionRate* depends on:

  - How many transactions actually update the replicated data
  - How many of those transactions are replicated through a particular outbound queue

  For example, if the *InboundTransactionRate* is 50 transactions per second, and half of those transactions update replicated tables, and 20 percent of the updates to the replicated tables are replicated through queue 1, the *OutboundTransactionRate* for queue 1 is 5 transactions per second (50 * 0.5 * 0.2).

  Calculating the *OutboundTransactionRate* can be complicated if transactions contain updates to many different replicated tables with different replication selectivities. In such

cases, the *InboundTransactionRate* provides a convenient upper bound for the *OutboundTransactionRate*, and you may use it in your formulas instead of the *OutboundTransactionRate*.

If all transactions in your database are replicated through an outbound queue, then the *OutboundTransactionRate* is the same as the *InboundTransactionRate*.

### Database Volume

Calculate a rough estimate of the upper limit of database volume by adding together the upper bound message rates for each table.

```
InboundDatabaseVolumeupper = sum(InboundTableVolumeupper) +
InboundTransVolume
```

To more accurately determine the *InboundDatabaseVolume*, add together the table volumes that were calculated in "Table Volume Upper Bound Method."

```
InboundDatabaseVolume = sum(InboundTableVolume) +
InboundTransactionVolume
```

#### See also

- *Inbound Database Volume* on page 123
- *Table Volume Upper Bound Method* on page 115

### Inbound Queue Size

An inbound queue contains updates to all of the replicated tables in a primary database. The inbound queue keeps these updates for the duration of the longest open transaction. Queue sizes are expressed in bytes.

To calculate the average size of an inbound queue:

```
InboundQueueSizetypical = InboundDatabaseVolume *
TransactionDurationtypical
```

To calculate the maximum size of an inbound queue:

```
InboundQueueSizelongest = InboundDatabaseVolume *
TransactionDurationlongest
```

where:

- *InboundDatabaseVolume* is the volume of messages calculated by the formulas described in Database Volume.

  You can use the *OutboundDatabaseVolume* to calculate the *InboundQueueSize* if you do not need to be precise.
- $TransactionDuration_{typical}$ represents the number of seconds for a typical transaction.
- $TransactionDuration_{longest}$ represents the number of seconds for the absolute transaction.

In practice, the maximum size of the inbound queue is slightly longer than the calculation, because while a long transaction is being read, new transactions are being added to the log. Since the duration of the longest transaction is an approximation, when determining its value add an estimate for the short period required to read and process it from the stable queue.

Also note that the size of the queue may be larger if messages trickle slowly into the Replication Server. Replication Server writes messages to stable queues in fixed blocks of 16K. To reduce latency, it writes even partially full blocks every second. (You can use the **init_sqm_write_delay** configuration parameter to change the time delay to something other than 1 second.) If several messages arrive almost simultaneously, all of them are written into one block. But if the same messages arrive at staggered intervals, each may occupy separate block.

**Note:** Since Adaptive Server transaction log requirements may increase with new versions of Adaptive Server, Replication Server stable queue space requirements may also increase.

### See also

- *Inbound Queue Size Example Calculation* on page 123

### Outbound Queue Volume

An outbound queue sends data to another Replication Server or to a data server. In these calculations the destination is called a direct destination site.

For each database replicated to (or through) the direct destination site, some part of the table volumes passes through the queue. The fraction of a table replicated through a queue is called replication selectivity.

Calculate an upper bound of the *OutboundQueueVolume* by assuming the highest possible replication selectivity of 1 (or 100 percent) for all tables. Then, add together *OutboundTableVolumes* for all tables replicated through the queue.

```
OutboundQueueVolumeupper= sum(OutboundTableVolumes) +
OutboundTransactionVolume
```

For example, if two tables with *OutboundTableVolumes* of 20K per second and 10K per second and an assumed *OutboundTransactionVolume* of 1K per second are replicated through an outbound queue, the *OutboundQueueVolume* would be:

```
20K/Sec + 10K/Sec + 1K/Sec = 31K/Sec
```

You can obtain a more accurate measure of queue volume by factoring in a value for replication table selectivity. The formula is:

```
OutboundQueueVolume = sum(OutboundTableVolume *
ReplicationSelectivity) + OutboundTransactionVolume
```

For example, in the previous example, if only 50 percent of the first table was replicated and 80 percent of the second table was replicated, the *OutboundQueueVolume* is:

```
(20K/Sec*0.5) + (10K/Sec*0.8) + 1K/Sec = 19K/Sec
```

### See also

- *Outbound Queue Volume Example Calculation* on page 124

### Failure Duration

If the direct destination site is unavailable, the queue buffers its messages. The failure duration figure you use to establish queue size determines how long a failure the queue can withstand.

### Save Interval

The save interval specifies how long data is kept in outbound queues before it is deleted. The save interval helps in recovering from a disk or node loss.

Use the **configure connection** to specify the save interval for a database connection. Use the **configure route** to specify the save interval for a route to another Replication Server. See the *Replication Server Reference Manual* for additional information.

### Outbound Queue Size

Calculate the outbound queue size.

The size of an outbound queue is calculated using:

```
OutboundQueueSize = OutboundQueueVolume * (FailureDuration +
SaveInterval)
```

where:

- *OutboundQueueVolume* is the amount of data in bytes entering the queue per time period.
- *FailureDuration* is the maximum time the queue is expected to buffer data for an unavailable site.
- *SaveInterval* is the time the messages are kept in the queue before being deleted.

If an outbound queue with an *OutboundQueueVolume* of 18K per second has a *SaveInterval* of 1 hour, and is intended to withstand 1 hour of unavailability at the destination site, the size of the queue should be:

```
OutboundQueueSize =
 18K/sec * (60min + 60min) =
 0.018MB/sec * 120min * 60sec/min = 130MB
```

In practice, if the actual change rates and table selectivities result in less than a full 16K block being filled each second, then the figures obtain by these calculations are low. Replication Server writes a block every second, whether or not the block is full. The worst usage of queue space occurs when small transactions are executed more than a second apart. Statistics indicate that most blocks are from 50 percent to 95 percent full.

#### See also

- *Outbound Queue Size* on page 124

## Overall Queue Disk Usage

Calculate the overall disk space.

To calculate the overall total disk space required for worst-case queue usage, use the formula:

```
Sum[InboundQueueSize] + Sum[OutboundQueueSize]
```

where:

- *Sum[InboundQueueSize]* is the sum of the inbound queue sizes for each of the databases.
- *Sum[OutboundQueueSize]* is the sum of the outbound queue sizes for each direct destination.

## Additional Considerations

When planning your replication system, take the described considerations into account.

- When one of the remote sites is unavailable because of a network failure, it is likely that the other sites on the same network will be unavailable as well, so many queues will grow simultaneously. Thus, you need to allocate enough disk space for all queues to withstand a simultaneous failure of the planned-for duration.
- When the queues start filling up, you can always add more partitions if you have spare disk space.
- If all the queues fill up and the inbound queue cannot accept more messages, the primary database will not be able to truncate its log. If you manually override this restriction, replicate databases will not receive the truncated updates.

## Example Queue Usage Calculations

A series of sample calculations using the formulas for calculating the disk queue size.

### Calculation Parameters for Examples

Provides the calculated values derived by using the formulas for calculating the disk queue size.

These calculations use these assumptions about the example replication system:

- Tables `T1` and `T2` are in database `DB1`.
- Table `T3` is in database `DB2`.
- All three tables are replicated to sites S1, S2, and S3.
- Not all messages are replicated to all sites.

**Table 7. Table Parameters**

| Data-base | Table | # Cols | Columns Changed | Change Rate (num/ sec) | Row Width (bytes) | Row Width Changed | # Sites |
|---|---|---|---|---|---|---|---|
| DB1 | T1 | 10 | 5 | 20 | 200 | 100 | 3 |
| DB1 | T2 | 10 | 5 | 10 | 400 | 200 | 2 |
| DB2 | T3 | 120 | 100 | 2 | 1500 | 1000 | 2 |

**Table 8. Site Parameters—Table Replication Selectivity**

| Table | Percent of Updates to S1 | Percent of Updates to S2 | Percent of Updates to S3 |
|---|---|---|---|
| T1 | 10% | 40% | 40% |
| T2 | 40% | 80% | 0% |
| T3 | 20% | 0% | 20% |

**Table 9. Transaction Rates**

| Transaction Rate | *DB1* | *DB2* | S1 | S2 | S3 |
|---|---|---|---|---|---|
| Inbound | 20/sec | 20/sec | - | - | - |
| Outbound | - | - | 5/sec | 10/sec | 8/sec |

## Message Size Example Calculations

Sample calculations using the formulas for message sizes.

These examples use the *RowWidthChanged* formula (without minimal columns) for the calculation.

### See also

- *Message Sizes* on page 126

### Table Update Calculations

An example that estimates the upper limit of message size.

Using the following formulas:

```
InboundMsgSizeupdate = InboundMsgOverhead +
  ColOverhead + RowWidth + RowWidthChanged
```

```
OutboundMsgSizeupdate = OutboundMsgOverhead +
  RowWidth + RowWidthChanged + (NumSites*8)
```

The calculations for updates for each site are:

```
InboundMsgSizeT1 = 380 + 450 + 200 + 100 = 1130
  bytes
```

```
InboundMsgSizeT2 = 380 + 450 + 400 + 200 = 1430
  bytes
```

```
InboundMsgSizeT3 = 380 + 6600 + 1500 + 1000 = 9480
  bytes
```

```
OutboundMsgSizeT1 = 200 + 200 + 100 + 24 = 524 bytes
```

```
OutboundMsgSizeT2 = 200 + 400 + 200 + 16 = 816 bytes
```

```
OutboundMsgSizeT3 = 200 + 1500 + 1000 + 16 = 2716
  bytes
```

### Begin/Commit Pairs

Provides the values by using the formula to calculate message sizes for begins and commits

```
MsgSizebegin = 250
```

```
MsgSizecommit = 200
```

## Change Rate

The change rate is the maximum number of data modifications performed on a table per time period. The change rate differs for every table in your replication system.

## Table Volume Example Calculations

Calculates the table volumes for tables `T1`, `T2`, and `T3`.

For simplicity, what is presented here is a worst-case analysis that assumes all changes are due to SQL **update** statements plus their associated begin/commit pairs.

Using the upper bound formula for *InboundTableVolume*:

```
InboundTableVolume = (Max[InboundMsgSize]*ChangeRate)
```

The inbound table volumes for tables `T1`, `T2`, and `T3` are:

```
InboundTableVolumeT1 = 1130*20 = 23K/sec
```

```
InboundTableVolumeT2 = 1430*10 = 14K/sec
```

```
InboundTableVolumeT3 = 9480*2 = 19K/sec
```

Using the upper bound formula for *OutboundTableVolume*:

```
OutboundTableVolume = (Max[OutboundMsgSize]* ChangeRate)
```

The outbound table volumes for tables `T1`, `T2`, and `T3` are:

```
OutboundTableVolumeT1 = 524*20 = 10K/sec
```

```
OutboundTableVolumeT2 = 816*10 = 8K/sec
```

```
OutboundTableVolumeT3 = 2716*2 = 5K/sec
```

where:

- *InboundMsgSize* and *OutboundMsgSize* are taken from the calculations performed in Table Update Calculations topic.
- *ChangeRate* is taken from the *Table 7. Table Parameters* on page 120 .

### RSSD and Log Size Example Calculations

Each Replication Server has an RSSD with its own inbound queue. The RSSD also contributes to the queue volumes of the outbound queues.

However, since very little activity is expected in the RSSD, and all transactions are very small, you can assume that the disk space requirements for an RSSD are minimal:

- Inbound Queue = 2MB
- Outbound Queues = nothing

## Inbound Database Volume

An example that calculates the inbound database volume.

Based on the *InboundTable Volumes* calculated, you can calculate the *InboundDatabase Volume* using the upper bound formula:

```
InboundDataBaseVolume = sum(InboundTableVolume) +
InboundTransactionVolume
```

where:

- *Transaction Volume* is the sum of the *Message* $_{\text{begin}}$ and *Message* $_{\text{commit}}$ pairs (250 bytes + 200 bytes) times the *TransactionRate*.
- The transaction rates for each database are from the *Table 9. Transaction Rates* on page 121.

Thus, the *InboundDatabase Volumes* are:

```
InboundDatabaseVolumeDB1 = 23K + 14K +
 (450 bytes/tran * 20 tran/sec) = 46K/sec
```

```
InboundDatabaseVolumeDB2 = 19K +
 (450 bytes/tran * 20 tran/sec) = 28K/sec
```

## Inbound Queue Size Example Calculation

An example that calculates the inbound queue size.

Based on the *InboundDatabase Volumes*, you can calculate the maximum sizes of inbound queues for DB1 and DB2 using the formula:

```
InboundQueueSize = InboundDatabaseVolume * TransactionDuration
```

where:

- *Inbound Database Volume* is the volume of messages calculated.
- *TransactionDuration* is the number of seconds for the absolute longest transaction. For the purpose of this example, assume that the *TransactionDuration* for DB1 is 10 minutes, and for DB2 it is 5 minutes.

The *InboundQueue* sizes are:

```
InboundQueueDB1 = 46K/sec * 600sec = 28MB
```

```
InboundQueueDB2 = 28K/sec * 300sec = 8MB
```

### Outbound Queue Volume Example Calculation

Provides an example for calculating the outbound queue size.

Calculate the *OutboundQueueVolume* for tables `T1`, `T2`, and `T3`. Because selectivity is measured on a table basis (rather than a database basis), you must calculate outbound queue size for each table replicated through it. The formula is:

```
OutboundQueueVolume = sum(OutboundTableVolume *
ReplicationSelectivity) + OutboundTransactionVolume
```

where:

- *OutboundTableVolume* is taken from the calculations performed in Calculating Table Volume topic.
- *ReplicationSelectivity* values are from the *Table 8. Site Parameters—Table Replication Selectivity* on page 121.
- *TransactionVolume* is the size of the *Message* $_{\text{begin}}$ and *Message* $_{\text{commit}}$ pairs (250 bytes + 200 bytes) times the *TransactionRate*.

The formula for calculating the *OutboundQueueVolume* for Site 1 (S1) is:

```
OutboundQueueVolumeS1 = OutboundTransactionVolume +
(TableVolumeT1*ReplicationSelectivityT1,S1) +
(TableVolumeT2*ReplicationSelectivityT2,S1) +
(TableVolumeT3*ReplicationSelectivityT3,S1)
```

The *OutboundQueueVolumes* for the three sites are:

```
Site1 = (450*5) + (10K/sec*0.1) + (8K/sec*0.4) + (5K/sec*0.2) = 7K/
sec
```

```
Site2 = (450*10) + (10K/sec*0.4) + (8K/sec*0.8) + (5K/sec*0) = 15K/
sec
```

```
Site3 = (450*8) + (10K/sec*0.4) + (8K/sec*0) + (5K/sec*0.2) = 9K/sec
```

### Outbound Queue Size

Provides an example on how to calculate the outbound queue size.

The size of an outbound queue is calculated using:

```
OutboundQueueSize = OutboundQueueVolume * (FailureDuration +
SaveInterval)
```

where:

- *OutboundQueueVolume* is the amount of data in bytes entering the queue per time period.
- *FailureDuration* is the maximum time the queue is expected to buffer data for an unavailable site. For the purpose of these example calculations, assume that failure duration is set at 4 hours (14,400 seconds).

- *SaveInterval* is the time configured for that particular queue. For the purpose of these calculations, assume that the save interval is 2 hours (7200 seconds).
- *FailureDuration* + *SaveInterval* is 21,600 seconds.

The *OutboundQueueSizes* for the three sites are:

```
Site1 = 7K/sec * 21,600sec = 151MB
```

```
Site2 = 15K/sec * 21,600sec = 324MB
```

```
Site3 = 9K/sec * 21,600sec = 194MB
```

### Total Disk Queue Usage Example Calculation

Provides an example for calculating the total disk queue.

To calculate the total disk space necessary to handle all queues during a worst-case failure (4 hours), using:

```
Sum(InboundQueueSizes) + Sum(OutboundQueueSizes)
```

where:

- *InboundQueueSizes* are the two queues calculated in "Inbound queue size example calculation."
- *OutboundQueueSizes* are the three queues calculated above.

The total disk space needed in the worst case, including 2MB for the RSSD inbound queue, is:

```
2MB + 28MB + 8MB + 151MB + 324MB + 194MB = 707MB
```

Sybase recommends that you allocate enough space for your worst case scenario. If you use the save interval feature (outbound queues are not truncated even after messages are delivered to the next site), be sure to allocate enough queue space to sustain your peak transaction activity. If you do not use the save interval, then under normal circumstances, your queue utilization is very small, perhaps 1MB or 2MB per queue.

The example calculation assumes that all outbound queues must tolerate the same duration of failure. This assumption may not be true in your environment. Typically, connections across a WAN must tolerate longer duration failures than local connections.

#### See also

- *Inbound Queue Size Example Calculation* on page 123

## Message Sizes

A Replication Server distributes database modifications using messages in ASCII format. Most replication system messages correspond to the delete, insert, update, and function execution operations.

Each table has one message size for inserts or deletes and another for updates. Each function has its own message size. Message sizes are expressed in bytes.

Message sizes for the same type of modification (**insert**, **delete**, or **update**) may vary depending on whether the message is in the inbound or outbound queue.

There are formulas you can use to calculate message sizes in bytes for table updates, inserts, and deletes, for functions, and for begin/commit pairs.

### *Table Updates*

For a rough estimate of the upper limit of message size, use:

```
InboundMsgSizeupdate = InboundMsgOverhead +
  ColOverhead + (RowWidth*2)
```

```
OutboundMsgSizeupdate = OutboundMsgOverhead +
  (RowWidth*2) + (NumSites*8)
```

For a more precise estimate, use the *RowWidthChanged* figure in the calculation:

```
InboundMsgSizeupdate = InboundMsgOverhead +
  ColOverhead + RowWidth + RowWidthChanged
```

```
OutboundMsgSizeupdate = OutboundMsgOverhead +
  RowWidth + RowWidthChanged + (NumSites*8)
```

If you use the minimal columns feature, calculate message size using:

```
InboundMsgSizeupdate = InboundMsgOverhead +
  ColOverhead + (RowWidthChanged*2) +
  PrimaryKeyWidth
```

```
OutboundMsgSizeupdate = OutboundMsgOverhead +
  (RowWidthChanged*2) + PrimaryKeyWidth +
  (NumSites*8)
```

### *Table Inserts*

Calculate the message size for table inserts. This formula also applies if you use the minimal columns feature.

```
InboundMsgSizeinsert = InboundMsgOverhead +
  ColOverhead + RowWidth
```

```
OutboundMsgSizeinsert = OutboundMsgOverhead +
  RowWidth + (NumSites*8)
```

### *Table Deletes*

If you do not use minimal columns, calculate message size in table deletes using :

```
InboundMsgSizeinsert = InboundMsgOverhead +
  ColOverhead + RowWidth
```

```
OutboundMsgSizeinsert = OutboundMsgOverhead +
  RowWidth + (NumSites*8)
```

If you use minimal columns, calculate table deletes using:

```
InboundMsgSizedelete = InboundMsgOverhead +
  ColOverhead + PrimaryKeyWidth
```

```
OutboundMsgSizedelete = OutboundMsgOverhead +
  PrimaryKeyWidth + (NumSites*8)
```

### *Functions*

Calculate message size for functions:

```
InboundMsgSizefunction = InboundMsgOverhead +
  ParameterWidth +(RowWidth*2)
```

```
OutboundMsgSizefunction = OutboundMsgOverhead +
  ParameterWidth + (NumSites*8)
```

In the formula for inbound message size, *RowWidth* does not apply to the replicated functions feature because before and after images of replicated functions are not sent to the inbound queue.

### *Begin and Commit Pairs*

Calculate message sizes for begins and commits:

```
InboundMsgSizebegin = OutboundMsgSizebegin = 250
```

```
InboundMsgSizecommit = OutboundMsgSizecommit = 200
```

The total size of a begin/commit pair is 450 bytes. If typical transactions have many modifications, omit the begin or commit message sizes from your calculations; their contribution to overall message size is negligible.

### *Formula Components*

Component definitions for calculating data volume.

- *InboundMsgOverhead* equals 380 bytes. Each message includes a transaction ID, duplicate detection sequence numbers, and so on.
- *OutboundMsgOverhead* equals 200 bytes. Each message includes a transaction ID, duplicate detection sequence numbers, and so on.
- *ColOverhead*, which applies to the inbound queue only, equals 30 bytes of overhead per column:

  For an update operation without minimal columns:

  ```
  (NumColumns+NumColumnsChanged) * 30
  ```

For an update operation with minimal columns:

```
((NumColumnsChanged*2)+NumPrimaryKeyColumns) * 30
```

For an insert operation (with or without minimal columns):

```
NumColumns * 30
```

```
For a delete operation without minimal columns:
```

```
NumColumns * 30
```

For a delete operation with minimal columns:

```
NumPrimaryKeyColumns * 30
```

- *RowWidth* is the size of the ASCII representation of the table columns. For example: a `char(10)` column uses 10 bytes, a `binary(10)` column uses 22 bytes.

  For table updates, the *RowWidth* is multiplied by 2 because both before and after images of row are distributed to the replicate sites.
- *RowWidthChanged* is the width of the changed columns in the row. For example, you have 10 columns with a total *RowWidth* of 200 bytes. Half of the columns in the row change, giving you a *RowWidthChanged* measurement of approximately 100 bytes.
- *NumSites* is the number of sites a message is sent to. This matters only when small messages are distributed to many site and may be insignificant if you do not have many sites. You might want to omit the number of sites factor from the formulas that use it. In the formula, each site ID is 8 bytes in length, so *Numsites* is multiplied by 8.
- *ParameterWidth* is the size of the ASCII representation of the function parameters.
- *Begin/Commit Pair* is the combined size of the begin message header and the commit trailer, which equals 450 bytes.

### See also

- *Table Update Calculations* on page 121

# Other Disk Space Requirements

Details the space requirements for Replication Servers, Replication Agents, and data servers in a replication system.

## Stable Queues

When you install Replication Server, you set up a disk partition that Replication Server uses to establish stable queues.

### See also

- *Overall Queue Disk Usage* on page 119

## RSSD

Specifies the minimum RSSD disk space.

For the RSSD, allocate at least 10MB for the data and 10MB for the log. These replication system defaults are designed for a relatively small system. If you want sufficient space for hundreds of replication definitions and thousands of subscriptions, you increase the data and log space to 12MB.

Errors and rejected transactions are also placed in the RSSD. The administrator should periodically truncate the tables (`rs_exceptscmd`, `rs_exceptshdr`,and `rs_exceptslas`) that hold these errors and rejected transactions. If you dump stable queues to the RSSD to help diagnose problems, truncate those tables when they are no longer needed.

## ERSSD

Specifies the minimum ERSSD disk space.

For the Embedded RSSD (ERSSD), allocate at least 5MB for the data, 3MB for the log, and 12MB for the backup. These replication system defaults are designed for a relatively small system. If you want sufficient space for hundreds of replication definitions and thousands of subscriptions, increase your data and log space to approximately 25MB each.

Errors and rejected transactions are also placed in the ERSSD. The administrator should periodically truncate the tables that hold these errors and rejected transactions. If you dump stable queues to the ERSSD to help diagnose problems, truncate those tables when they are no longer needed.

## Logs

Specifies the minimum space requirement for log files.

Replication Servers write information, error messages, and trace output into their error log files. Allocate 1MB to 2MB of disk storage for these log files. If you are asked by Sybase Technical Support to turn on trace flags, you may need to make substantially more log space available.

RepAgent writes messages to the Adaptive Server error log file.

# Memory Usage

Replication Servers are separate processes in the replication system. RepAgent is an Adaptive Server thread.

## Replication Server Memory Requirements

Specifies the Replication Server memory requirements.

As a general guideline, on a Sun SPARC station, estimate that:

- A newly installed Replication Server uses about 7MB of memory for data and stacks.
- Each DSI connection adds about 500K. This value increases if you configure larger values for MD memory (**md_sqm_write_request_limit** configuration parameter).
- Every RepAgent connection adds 500K. This value increases if you configure larger values for SQT memory (**sqt_max_cache_size** configuration parameter).
- If you have thousands of subscriptions, or if you increase the function string cache size, you must account for that by adding more memory.
- Memory for subscription rules is a function of the columns referenced in the subscription and the number of rules. A typical subscription adds less than 80 bytes plus the combined size of all subscription predicate values.
- Function-string cache size can grow to 200K.
- Remaining system table cache size is based on the number of objects defined. Each replication definition consumes memory equal to approximately 250 times the number of its columns. This might be a factor if you have numerous replication definitions with many columns.

## RepAgent Memory Requirements

Specifies the RepAgent memory requirements, for overhead, schema cache, transaction cache, and text and image cache.

Most RepAgent memory comes from allocated Adaptive Server procedure cache (shared memory).

See the *Adaptive Server Enterprise System Administration Guide Volume 2* for information about increasing server memory.

### Overhead

Adaptive Server allocates 5612 bytes for each database for log transfer.

When RepAgent is enabled, Adaptive Server allocates an additional 2332 bytes of memory at start-up for each RepAgent.

### Schema Cache

The amount of memory used for the schema cache depends of number of objects (tables and stored procedures) and descriptors (columns and parameters) that are replicated. Each object and descriptor requires 128 bytes.

At start-up, Adaptive Server allocates 8K of memory—which is sufficient for 64 object or descriptors. Thereafter, memory is allocated in 2K chunks. Adaptive Server also allocates 2048 bytes for the hash table at start-up.

A “least recently used” (LRU) mechanism keeps the schema cache size manageable by removing from memory those objects/descriptors not recently referenced. Thus, RepAgent does not need enough memory to describe all replicated objects. At minimum, RepAgent needs enough memory to describe one replicated object.

### Transaction Cache

RepAgent requires 256 bytes for each open transaction. Transaction cache memory is allocated in 2K chunks. As transactions are committed or aborted, free memory is placed in a pool which must be used before new memory can be allocated.

RepAgent requires memory for the maximum number of open transactions. If sufficient memory is unavailable, RepAgent shuts down.

At start-up, Adaptive Server allocates 2048 bytes for a hash table.

### Text and Image Cache

Specifies the text and image cache size.

The `text` and `image` cache is not affected by the size of the `text` and `image` data. Rather, the amount of memory used is dependent on the number of replicated tables containing `text` and `image` data and the number of columns in the tables that contain `text` and `image` data. Each replicated table containing `text` and `image` data requires 170 bytes; each replicated column requires 52 bytes.

The `text` and `image` cache memory is allocated in 2K chunks. Memory is allocated only when replicated tables exist that contain `text` and `image` data.

The `text` and `image` cache uses a free memory pool and requires sufficient memory for all `text` and `image` data.

### Other Memory

Specifies other memory requirement for Replication Server.

- If RepAgent is enabled, Adaptive Server allocates one extra process descriptor for RepAgent.
- RepAgent uses Client-Library to connect to Replication Server. **ct-lib** allocates memory directly (dynamic memory) as needed.

## CPU Usage

Replication Server runs on multiprocessor or single-processor platforms. The Replication Server multithreaded architecture supports both hardware configurations.

When Replication Server is configured with the symmetric multiprocessor (SMP) feature turned off, Replication Server threads run serially. A server-wide mutually exclusive lock (mutex) enforces serial thread execution, ensuring that threads do not run concurrently on different processors.

When Replication Server is configured with the SMP feature turned on, Replication Server threads can run in parallel, thereby improving performance and efficiency. The server-wide mutex is disengaged and individual threads use a combination of thread management techniques to ensure that global data, server code, and system routines remain secure.

To enable SMP on a multiprocessor machine, use **configure replication server** with the **smp_enable** option. For example:

```
configure replication server set smp_enable to 'on'
```

Replication Server support for multiple processors is based on Open Server support for multiple processors, that is, a single process running multiple threads. Replication Server uses the POSIX thread library on UNIX platforms and the WIN32 thread library on Windows platforms. For detailed information about Open Server support for multiple processor machines, see the *Open Server Server-Library/C Reference Manual*.

## Network Requirements

Specifies the network requirements for Replication Server.

When you have calculated the insert rates into the outbound queues, you can get an idea of the network throughput required. The throughput should allow a Replication Server to send messages faster than messages are being added to the queues. Network messages are usually fractionally larger than what is written to the outbound queues.

Sometimes the queues are filled in bursts and are slowly drained through a potentially low bandwidth network. Consider this behavior when you calculate queue sizes, and make sure that your queues can handle bursts of incoming data in addition to connection and destination site failures. In addition, collect monitor and counter data during different work-loads to help you accurately calculate queue sizes.

# Index

## A



## B



## C

## D

## E



## F



## G



## H



## I



## J



## L

## M



## N



## O



## P

## R

## S



## T

## U



## V



## W